3-2004

An Analysis of the Performance and Security of J2SDK 1.4 JSSE Implementation of SSL/TLS

Danny R. Bias

AN ANALYSIS OF THE PERFORMANCE AND SECURITY OF J2SDK 1.4 JSSE IMPLEMENTATION OF SSL/TLS

THESIS

Danny R. Bias, Captain, USAF

AFIT/GCS/ENG/04-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

AFIT/GCS/ENG-04-02

AN ANALYSIS OF THE PERFORMANCE AND SECURITY OF J2SDK 1.4 JSSE IMPLEMENTATION OF SSL/TLS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Danny R. Bias, BS

Captain, USAF

March 2004

AFIT/GCS/ENG-04-02

AN ANALYSIS OF THE PERFORMANCE AND SECURITY OF J2SDK 1.4 JSSE IMPLEMENTATION OF SSL/TLS

Danny R. Bias, BS

Captain, USAF

Approved:

// SIGNED //	11 Mar 2004
Dr. Richard A. Raines, DAF (Chairman)	Date
// SIGNED //	10 Mar 2004
Maj. Rusty O. Baldwin, PhD (Member)	Date
// SIGNED //	10 Mar 2004
Dr. Gilbert L. Peterson, DAF (Member)	Date

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Dr. Richard Raines, for his guidance and support throughout the course of this thesis effort. The insight and experience was certainly appreciated. I would, also, like to thank my sponsor, Mr. Neal Ziring, from the National Security Agency for both the support and latitude provided to me in this endeavor.

Danny R. Bias

Table of Contents

Page

List of Figures

Page

List of Tables

Page

Abstract

The Java SSL/TLS package distributed with the J2SE 1.4.2 runtime is a Java implementation of the SSLv3 and TLSv1 protocols. Java-based web services and other systems deployed by the DoD will depend on this implementation to provide confidentiality, integrity, and authentication. Security and performance assessment of this implementation is critical given the proliferation of web services within DoD channels. This research assessed the performance of the J2SE 1.4.2 SSL and TLS implementations, paying particular attention to identifying performance limitations given a very secure configuration.

The performance metrics of this research were CPU utilization, network bandwidth, memory, and maximum number of secure socket that could be created given various factors. This research determined an integral performance relationship between the memory heap size and the encryption algorithm used. By changing the default heap size setting of the Java Virtual Machine from 64 MB to 256 MB and using the symmetric encryption algorithm of AES256, a high performance, highly secure SSL configuration is achievable. This configuration can support over 2000 simultaneous secure sockets with various encrypted data sizes. This yields a 200 percent increase in performance over the default configuration, while providing the additional security of 256-bit symmetric key encryption to the application data.

AFIT/GCS/ENG-04-02

AN ANALYSIS OF THE PERFORMANCE AND SECURITY OF J2SDK 1.4 JSSE IMPLEMENTATION OF SSL/TLS

1. Introduction

The Java SSL/TLS package distributed with the J2SE 1.4.2 runtime is a Java implementation of the SSLv3 and TLSv1 protocols. Java-based web services and other systems deployed by the DoD will depend on this implementation to provide confidentiality, integrity, and authentication. Security and performance assessment of this implementation is critical given the proliferation of web services within DoD channels. This thesis is organized into five distinct chapters; introduction, background, methodology, results and analysis, and the conclusion.

The background chapter sets up the research effort by giving detailed background on Java and how it has evolved to become a secure architecture. The Secure Sockets Layer (SSL) protocol is also explained since JSSE is just an implementation of this well documented standard. The handshake and record protocol within SSL is mapped to show how the SSL process of encryption actually works. Specifics of how JSSE implements SSL are given and a mapping is shown of all the Java classes used in the implementation. Security concerns of any SSL implementation include:

- Malformed SSL communications which could cause incorrect ciphers to be used
- Timing and cryptographic attacks on SSL
- Padding attacks

- Denial of Service and resource exhaustion

Chapter Three, the methodology chapter starts with the statement of the problem the research is addressing as well as detailing the four goals of the research and the hypotheses. After the goals are defined, the boundaries of the system are laid out and defined. The services provided by the system are documented and performance metrics are identified. Parameters are identified and their static values are determined. The workload to be used in the simulation program is defined and explained as are the factors that will determine how many experiments are going to be necessary to test the system. The evaluation technique is explained along with code segments used in the development of the simulation program. Specific attention is focused on how the factors and parameters are implemented. The chapter concludes with an explanation and details of the experimental design.

Chapter Four focuses on the analysis and results of the experimentation. The raw data obtained from the simulation scenarios are presented as well as the statistical techniques used to analyze the data. The raw data considered vital to the presentation were appended to the paper as Appendices A and B. The analysis of the data is explained and the answers are matched to the investigative questions of the research effort.

The final chapter discusses the conclusions of the research and the significance of the effort. Based upon the conclusions reached from the study, recommendations for action are detailed for setting up a JSSE implementation. Since this research was unable to touch all associated areas, there are also recommendations for future research in this area.

2. Background

This chapter provides background information on the research topic. In it, the history of security in Java is discussed as well as the basics of the Secure Sockets Layer (SSL) protocol. These are intended to explain why SSL was incorporated into Java via the Java Secure Sockets Extension (JSSE).

2.1 Introduction

The next two sections explain the current Java Security Architecture and the JSSE implementation of SSL. These sections provide information and sources on how SSL was implemented into Java.

The final parts of the chapter provide background information on certain attacks to which the Java implementation of SSL may be susceptible. Denials of Service and resource exhaustion possibilities are included. Information on the tools of the research effort and a summary concludes the chapter.

2.2 History of Security in Java

2.2.1 Mobile Code

The big advantage of Java is its portability. Through the use of Java applets, developers are able to create code that can be downloaded directly into a Web browser. This technology is one of the first that turned the Web browser into a framework that could support the execution of applications downloaded over the Web [Pec00]. This creates a new paradigm for computing, which is in stark contrast to traditional desktop computing.

With traditional desktop computing, applications are loaded and executed by the user on a local machine. Whenever updates to application software are needed, updates are obtained from sources such as CD, removable disks, and tapes. The updates are then manually loaded. Java applets are a new paradigm in which mobile code is downloaded dynamically to a local Web browser and automatically updated whenever a revisit to the Web site from where the code was downloaded is made [McG02].

This grand vision is somewhat tempered by the fact that network data rates are not where they need to be for large updates [Pec00]. This limits the realistic size of the Java applets and therefore limits the complexity of the applications downloaded. Another limiting factor is the performance of the Java Virtual Machine (JVM) implementations equipped with Web browsers.

Despite the problems associated with mobile code provided by Java applets, Sun Microsystems did realize there would be some implementation of these applets. They further understood that for applets to be viable for business and government use, they would need security designed in at an early stage. It is believed that most users want to limit access to their local machines when downloading software from remote Web sites. Many traditional desktop applications require access to the local file system, but with increased security risks due to malicious software, this access needs to be limited. Java security developers have tried to keep pace with what seems to be critical at a given time. Java security has gone through several different iterations to keep pace with the current Web security environment [Pec00].

2.2.2 Java 1.0 Sandbox Security Model

The Java security model has evolved with each major Java version release. Java version 1.0 platform provided a very limited security model known as the "sandbox" model. In the sandbox model, only local code had access to all the resources (files, new network connections, etc.,) that were exposed and accessed by the JVM. Code downloaded from remote sources such as applets only had access to a limited number of resources. Thus, file-system access and the capability to create new connections were limited for remote code. This was a major concern for JVM implementations equipped with Web browsers.

The Java 1.0 security model was too restrictive and did not allow much flexibility for Java developers. The ability to provide downloadable applications over the Web was being stifled by the fact that such applications could not perform key operations such as file access or create new network connections. If Web-browser vendors treated remote code like local code, the path would have been opened for malicious code to corrupt the local machine. Such an all-or-none model was replaced in Java 1.1, when a trusted security model was employed [Pec00].

2.2.3 Java 1.1 Trusted Code Security Model

With the trusted code model, the user can optionally designate whether code "signed" by certain providers is allowed to have the full resource access it desires. Thus, a trust relationship can be established for Microsoft Java code to run inside of your browser with full access to system resources much like the trust that exists when one of many Microsoft products is installed on the local system. Code or applet signing permits

a company like Microsoft to sign its applet so that the origin of the code can be verified. The signed applet grants access to all system resources, much like the traditional desktop method. Untrusted code can still be confined to the sandbox, since each individual applet is treated with a discrete set of rules.

2.2.4 Java 1.2 Configurable and Fine-Grained Access Security Model

The Java 2 platform (also called Java 1.2) has much finer-grained application security. With this new model, local and remote code alike can be confined to use only particular domains of resources according to configurable policies. Consider, for example a Java code segment called Foo. Foo may have limited access to resources which are confined within a single domain (defined via the Java application's access control list). Some other Java code, Bar, may have access to a set of resources confined by some other domain [Pec00]. Domains of access and configurable security policies make the Java 2 platform much more flexible. This design abstracted the distinction between remote and local code, allowing developers to focus on a wider range of security problems (secrecy, authenticity, integrity, etc.), instead of focusing on the mobile code and Java applet security problems. Secure Sockets Layer (SSL) was an obvious choice to take care of some of these other security problems. Before discussing Java implementation of SSL, a quick look at SSL basics is in order.

2.3 Secure Sockets Layer (SSL) Basics

2.3.1 SSL Protocol

The Secure Sockets Layer (SSL) protocol is the most widely used security protocol for authentication on the Internet [ViM02]. It secures data exchanged between a client and a server by encrypting it. In general, it provides three of the tenants of data security: authentication, integrity, and confidentiality. Authentication is the process of ensuring the "real" parties wishing to communicate do so and are not fooled by an entity impersonating an identity. Authentication is achieved through the use of asymmetric public key encryption. Integrity guarantees data exchanged with the server has not been modified along the way. If it is, it can be detected through the use of the Message Authentication Code (MAC) [ViM02]. The MAC is generated during the SSL Handshake through a pseudo-random number generator and a secure hash algorithm. Finally, confidentiality is achieved through data encryption. An eavesdropper cannot read the transmitted information by simply looking at the packets on the network. The following sections give more detail about the two protocols that make up the Secure Sockets Layer protocol.

a. Handshake Protocol

The SSL HandShake Protocol is the most complex part of SSL. It enables a server and client to authenticate each other, as well as negotiating encryption, MAC algorithm, and cryptographic keys. It is used before any application data is sent. Figure 1 shows a Handshake Protocol session.

SSL HANDSHAKE PROTOCOL

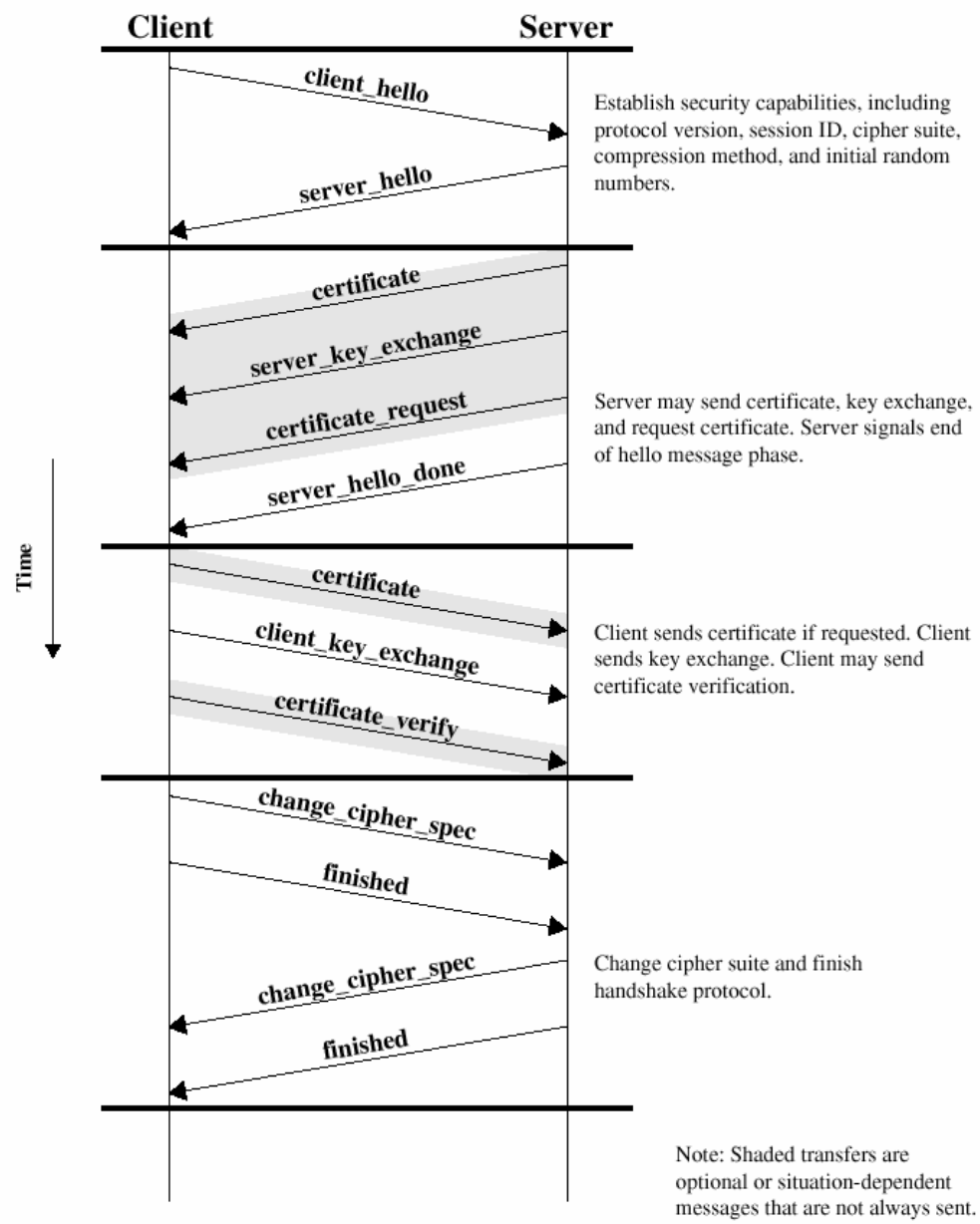

Figure 1: SSL Handshake Protocol

b. Record Protocol

When data is being transmitted, the Record Protocol receives unencrypted data from the higher layer (Handshake Protocol) and changes it into SSL

CipherText. The record layer fragments information blocks into TLSPlaintext records containing 2^{14} bytes or less of data. Client message boundaries are not preserved in the record layer (i.e., multiple client messages of the same ContentType may be coalesced into a single TLSPlaintext record, or a single message may be fragmented across several records) [RFC2246]. It compresses the data and appends the Message Authentication Code (MAC). Finally, before transmitting the data, it is encrypted with a shared symmetric key.

If data is being received, the Record Protocol receives SSL CipherText from the TCP layer and decrypts using the shared symmetric key. It uses the MAC to verify the integrity of the message and decompress the data. Before sending the unencrypted data to the higher layers, it reassembles the fragmented data. Figure 2 depicts a Record Protocol session.

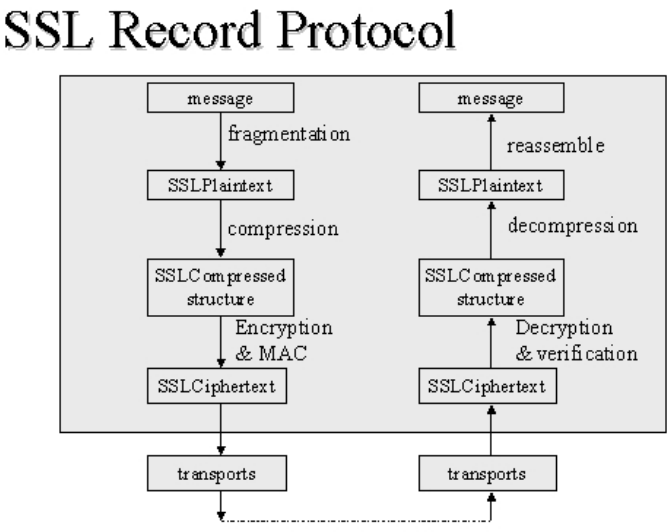

Figure 2: SSL Record Protocol

2.3.2 Cipher Suites

Cipher Suites are the building blocks SSL uses to provide authentication, integrity, and secrecy. They are used in both of the SSL protocols and are selected based upon the handshake between the server and the client. Since not all clients and servers are the same, SSL supports many different types of cipher suites.

The two different SSL cryptographic algorithms, asymmetric and symmetric, use a combination of the available cipher suites to provide secure communications between the client and the server. Which suite is selected is based upon the sensitivity of the data involved, the speed of the cipher, and the applicability of export restrictions on the stronger encryption algorithms.

The asymmetric algorithm is used in the handshake protocol to authenticate the different parties, generate shared keys and secrets. Asymmetric algorithms use much more complex numbers than do symmetric algorithms and are generally slower to process [ViM02]. These algorithms use a public/private key combination for authentication. The public key can be known by anyone. Only the owner knows the private key. The public key is used by the client and server to encrypt and decrypt the premaster (used to generate the smaller shared symmetric key). The public key is an inverse prime of the private key. This inverse property allows the server to decrypt a message encrypted with the corresponding public key.

Symmetric algorithms are used to encrypt and verify the integrity of SSL records (application data). A symmetric algorithm is used for the large data streams due to the speed with which the data can be encrypted and unencrypted. Speed is critical when

maintaining a high traffic server. The symmetric key is passed between the client and the server via the asymmetric method.

Listed below are some of the cipher suites offered in most SSL implementations.

a. DES – Data Encryption Standard [NIST99]
b. DSA – Digital Signature Algorithm [NIST00]
c. KEA – Key Exchange Algorithm [NIST98]
d. MD5 – Message Digest algorithm developed by Rivest [RFC1321]
e. RC2 and RC4 – Rivest ciphers developed for RSA Data Security [RFC2268]
f. RSA – A public-key algorithm for both encryption and authentication. Developed by Rivest, Shamir, and Adleman [RFC2437]
g. RSA key exchange – A key exchange algorithm for SSL based on the RSA algorithm [NIST00]
h. SHA-1 – Secure Hash Algorithm, developed by the U.S. Government [NIST95]
i. Triple-DES – DES applied 3 times [NIST99]

2.4 Java Security Architecture

The Java security architecture is made up of three different parts. These parts are the Core Java Security Architecture which contains the basic security capabilities, the Java Cryptography Architecture (JCA) which allows basic cryptography functionality for Java applications, and the Java Security Extensions (JSE) which allow various third party vendor implementations via standard interfaces such as SSL.

2.4.1 Core Java 2.0 Security Architecture

The Core Java 2.0 Security Architecture has seven components. The following is a short description of each. The *byte code verifier* verifies that the byte codes being loaded from Java application code external to the Java platform adhere to the syntax of the Java language specification. The *class loader* is responsible for actual translation of byte codes into Java class constructs that can be manipulated by the Java runtime environment. In the process of loading classes, different class loaders may employ different policies to determine whether certain classes should even be loaded into the runtime environment. The class loader and the Java 2 platform classes limit access to valued resources by intercepting calls made to Java platform APIs and delegating decisions as to whether such calls can be made to the security manager. Java 1.0 and 1.1 made exclusive use of a security manager for such decision making, whereas Java 2 applications uses the access controller for more flexible and configurable access control decision making. Finally, execution of code would not be possible without the runtime execution engine. Access control was a significant addition to the Java 2 security architecture. It extended the security model to allow configurable and fine-grained access control. Java 2 permissions have configurable and extendable ways to designate access limitations and can be associated with valued resources. Java policies provide the mechanisms needed to actually associate such permissions with valued resources in a configurable way. Finally, the ability to encapsulate domains for access control are provided with the core Java 2 security model [PeC00].

2.4.2 Java Cryptography Architecture

The Java Cryptography Architecture (JCA) provides an infrastructure for performing basic cryptographic functionality with the Java platform. The scope of cryptographic functionality includes protecting data against corruption using basic cryptographic functions and algorithms. Cryptographic signature generation algorithms used for identifying sources of data and code are also built into the JCA. Because keys and certificates are a core part of identifying data and code sources, APIs are also built into the JCA for handling such features.

Even though the JCA is part of the built-in Java security packages as defined in the core Java 2 security architecture features, it is separate due to the JCA's underlying service provider interface. Different cryptographic implementations can be plugged into the JCA framework without affecting the Java applications. The Object Oriented method by which the security framework is built allows this. For developers who are not sure or do not care what cryptographic functions are used, there is a default set of cryptographic functions that are instantiated when JCA is used.

2.4.3 Extensions

a. Java Cryptography Extension

The terms encryption and cryptography are sometimes used interchangeably. However, Sun Microsystems Inc. (referred to hereafter as Sun) adheres to a cryptographic definition that includes basic data integrity and source identity functions supported by the JCA. Encryption are those functions used to encrypt blocks of data for the added sake of confidentiality until the data can be

subsequently decrypted by the intended receiver. The Java Cryptography Extension (JCE) is provided as a Java security extension for these auxiliary encryption purposes. It could be argued that encryption is a core aspect of any secure system. However, Sun has purposely made JCE an extension to the Java architecture largely due to U.S. export restrictions on encryption technology. If JCE was a core part of the Java architecture, exportability of the Java architecture itself would be hampered. Although many commercial-grade encryption technologies have been developed by third parties, JCE includes a standard service provider and application programmer interface model. Thus, different commercial-grade encryption implementations can be used and still provide the programmer with the same API to the different underlying implementations. Another difference between the JCE and JCA is that the JCA primarily supports data protection for integrity via message digests and provides a means for identification of data, objects, and code using signatures, keys, and certificates. The data, objects, and code are never encrypted. This is part of JCE. Where JCA relies on asymmetric public and private key infrastructure for secure identity, JCE relies on a symmetric key infrastructure for confidentiality.

b. Java Secure Socket Extension (JSSE)

It is important to describe the JSSE in context where it fits into the Java Architecture. Since SSL is one of the more commonly used encryption-based protocols for integrity and confidentiality, Sun developed the JSSE as an extension to the Java security architecture. JSSE provides a standard interface

along with an underlying reference implementation for building Java applications with SSL. Different commercial-grade SSL implementations can be used with JSSE and still provide the same interface to the applications developer. This has advantages and disadvantages since some SSL implementations may be more prone to attack than others. Some of these attacks are discussed later. JSSE is more generic to provide a standard interface to support other secure socket protocols such as the Transport Layer Security (TLS) and Wireless Transport Layer Security (WTLS) protocols.

c. Java Authentication and Authorization Service

The Java Authentication and Authorization Service (JAAS) extension to the Java security architecture was developed to provide a standard way to limit access to resources based on an authenticated user identity. Thus, standard APIs for login and logout are provided such that a standard interface is available for passing around secure user credentials and context. This makes it possible to swap in and out different underlying authentication model implementations.

2.5 JSSE Implementation of SSL

There is little open-literature on specific instances of where JSSE's implementation is significantly different than that of most other implementations. Building bad cryptographic systems is very easy to do, while building strong cryptographic systems is very hard and time consuming [Sch98]. Sun chose to use a cryptographic implementation that has been improved upon for a number of years and is now an accepted standard of the Internet Engineering Task Force. This section explains the key

classes used in the JSSE Application Program Interface (API). Keeping that in mind, a brief examination of the major class relationships for a JSSE SSL connection is given.

For secure communication, both ends of an SSL connection (server and client) must be SSL-enabled. Within the JSSE API, the endpoint class of the connection, and one of the core classes, is the SSLSocket. In Figure 3 below, the major classes used to create SSLSockets are laid out in a logical ordering [Sun03].

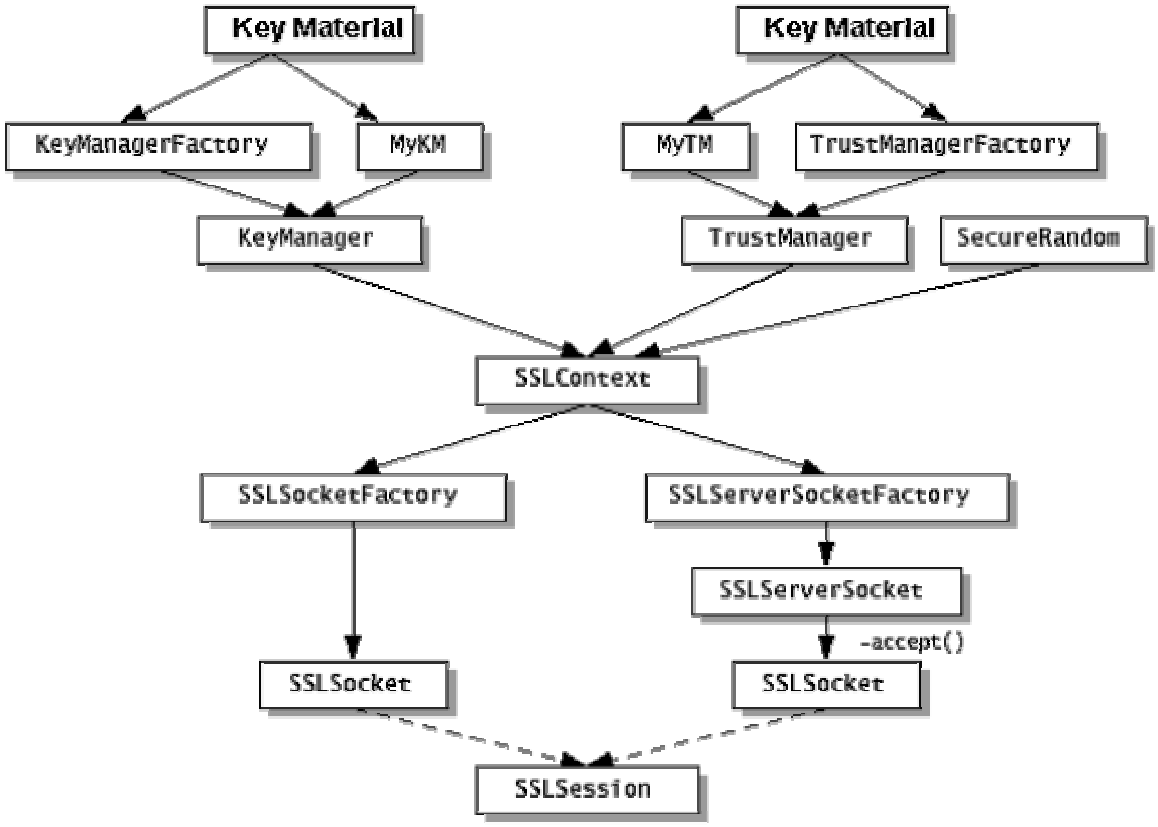

Figure 3: JSSE classes used to create SSLSockets [Sun03].

An SSLSocket is created by either an SSLSocketFactory or an SSLServerSocket accepting an in-bound connection (See Appendix C for a description of the classes and functions). Both SSLSocketFactory and SSLServerSocketFactory objects are created by an SSLContext. Additionally, there are two ways to obtain and initialize an SSLContext.

The simplest is to call the getDefault method on either of the Factory classes. This causes a default KeyManager, TrustManager, and a secure random number generator to be created. This process uses key material found in the default keystore as determined by system properties described in "Customizing the Default Key and Trust Stores, Store Types, and Store Passwords" section of the JSSE Reference Guide for Java 2 SDK Standard Edition V1.4.2 [Sun03].

Another approach that gives the caller the most control over the behavior of the created SSLContext, is to call the static method *getInstance* on the SSLContext class, and initialize the context by calling the *init* method. This method takes three arguments (array of KeyManager objects, array of TrustManager objects, and a SecureRandom random number generator. This allows the caller to dictate what cipher suites are allowed when creating the SSLSession. For instance, DES is a valid algorithm (56 bit key) but not very strong due to its short key length. DES3 is a more secure solution to implement on the session, though not necessarily the most efficient.

Although key length is important, it is actually a minor player in building secure cryptographic systems. It is a piece of the puzzle, but even a 168-bit key (DES3) does not protect the data if other parts of the cryptographic system itself are inherently flawed [Sch98]. There are many avenues of attack on a network connection [Rei96]. The next few sections look at some other ways to attack a cryptographic system.

2.6 Malformed SSL Communications

This section covers items of interest that should be validated during the testing phase of any application. The valid fields and ranges associated with the SSLSession communication between the client and the server are explained below.

2.6.1 Malformed SSL Messages.

Earlier sections described how SSL communications occur at the TCP and SSL protocol levels. Software can store input values or the results of computations internally in one or more data structures to be retrieved or passed for use in computation or output generation. Any type of software is set up for failure if it stores illegal data. So, care must be taken to keep the data structures free of such corruption [Whi03]. The two major areas to focus on for SSL are the legal ranges of the fields within the protocols that make up SSL. We focus on the two that are most used; the Record and Handshake Protocols.

2.6.2 Correctness of Cipher Suite Handling

Another area of concern within SSL communications is the correctness of the cipher suites. Each side of an SSL session is supposed to be able to have a list of preferred cipher suites and what order they appear. A couple of questions need to be answered to determine if there are weaknesses in this area for the JSSE implementation. Does JSSE implement this list correctly? Some older SSL implementations did not, and thus allowed an unscrupulous server to force a client to use a weaker cipher suite than it should. Also, does JSSE allow this kind of attack?

2.7 Timing and Cryptographic Attacks on SSL

This section discusses additional mechanisms hackers may use to break SSL encryption. Specifically, padding attacks, timing attacks, and random number generation are discussed. These attacks are not aimed at finding weaknesses within an encryption algorithm. At this point, the hacker is assuming the proverbial front door is locked. Now, the hunt is for weaknesses in the implementation of the cryptography. This is analogous to using a six inch steel front door only to leave all windows opened.

2.7.1 Padding Attacks

Padding is used in encryption algorithms to hide the length of the actual data and prevent certain dictionary; replay/déjà vu attacks on encrypted data. One thing cryptographers have to do is make sure any fix or patch put in place does not create additional problems. Unfortunately, this is difficult to do. This opens the system for padding attacks. Basically, this can occur if an attacker knows the algorithm used to institute the padding (Secure Hash Algorithm (SHA), Message Digest-5 (MD5), etc.,) and the algorithm is not sufficiently random or strong. The attacker will find a way to derive important data about the cleartext or even about the factorization of N, where N is the RSA modulus [Nac99]. With the factorization of N, the public and private keys used to encrypt and decrypt the message can be derived. The Java 2 Standard Edition 1.4.1 implementation of the Java Cryptography Extension has several different options to choose from. They include:

1. No padding
2. Optimal Asymmetric Encryption Padding

3. PKCS5Padding – Password-Based Encryption Standard
4. SSL3Padding – defined in SSL Protocol Version 3.0

This last option is offered in SunJCE but is not supported. None of these padding algorithms, except "no padding", are known to have any inherent flaws.

Another example of a padding attack, the Bleichenbacher attack isn't really a true padding attack [BoB03]. This attack takes advantage of a poor implementation of padding. In PKCS version 1, the padding algorithm added a 16-bit "02" to the beginning of the packet. If we know the first two characters, this is usually enough to break the encryption. For example, suppose Ivan intercepts a ciphertext C intended for Bob and wants to decrypt it. To mount the attack, Ivan picks a random r in the set Z from 1 – N, computes C' = rC mod N, and sends C' to Bob's machine. An application running on Bob's machine receives C' and attempts to decrypt it. It either responds with an error message or doesn't respond at all . This allows Ivan to learn whether the most significant 16 bits of the decryption of C' are equal to "02". In effect, Ivan has an oracle that tests for him whether the 16 most significant bits of the decryption of rC mod N are equal to 02, for any r of his choosing. Bleichenbacher showed that such an oracle is sufficient for decrypting C [Bon99].

2.7.2 SSL timing

Timing attacks are simple in concept, but difficult to implement in a wide area network. They basically send an encrypted message back to the target system. But, instead of using the proper public key, they encrypt the message with the attacker's guess of q where N is the RSA modulus and N = pq where p and q are sufficiently large primes

with q < p [Bob03]. The attack measures how much time it takes the target system to compute the fact that the message was encrypted with the wrong key. This attack basically discovers "q" one bit at a time based upon the time it takes to receive an error response back from the target system.

There are mechanisms that can be implemented to deny this activity. One way and probably the simplest is to set some random wait time before calculating the key. Another more elegant way is called *RSA hiding*. This techniques picks some random number r, where r is in the set of numbers 1 – N, e is the public key, d is the private key, C is enciphered M, and M is the original plain text message. It then calculates $C' = C * r^e \bmod N$. Now it will apply d to C' and obtains $M' = C'^d \bmod N$. At the last step, the original message is recovered by setting $M = M'/r \bmod N$. With this approach the target system is applying the private key to a random message derived from r. Since there is no knowledge of the bitstream that d was applied to, the attacker cannot gain an advantage by measuring the time. Not all implementations of SSL use RSA hiding. At the present, it is unknown if JSSE utilizes it.

2.7.3 Random Number Generation

Practical cryptography relies on the availability of a good source of randomness or at least a good source of pseudo-randomness. In Sun's JDK, the java.security.SecureRandom class provides security-grade pseudo-randomness.

Sun's implementation of the default provider for this class uses the SHA1 algorithm as the foundation for its random number generation. It seeds the pseudo-random number generator with a random value derived from thread timings.

2.8 Denial of Service and Resource Exhaustion Possibilities

This area of the thesis investigates how well Java handles resource exhaustion or denial of service attempts. To determine the performance, the number of JSSE SSL sessions allowed before system failure is examined. Additionally, a virtually identical OpenSSL server implementation will be used for determine possible differences in operational execution speeds resulting from different implementations. Each implementation will be examined for modifiable are to determine the consumption of resources and the overall impact on security performance that may result.

2.9 Tools

The tools use in this research include: Windows perfmon utility, UNIX Top utility, Ethereal protocol analyzer, JMP statistical package, Microsoft Excel, and Turbo C. The performance monitor utility in the Windows operating system provides the ability to record the application parameters such as memory and CPU utilization and then display them online in graph and report views. It was used in this research to gather server performance data. The UNIX Top utility gathers memory and CPU utilization data on UNIX systems in a variety of formats. For this research, it was used to gather data on the Linux server. Ethereal was used to verify data encryption, validate the correct algorithm was selected, and to gather data on the round trip time for secure socket data. JMP is an interactive software tool especially designed for statistical visualization and exploratory data analysis. Microsoft Excel was used to analyze the data from the Top and Perfmon utilities, and Turbo C was the programming language used to code various data

conversion routines to convert the data from the Top utility to a format to be used in the Excel analysis.

2.10 Summary

This chapter provides background information on the research topic. In it, the history of security in Java is discussed, as well as the basics of the Secure Sockets Layer (SSL) protocol. Its intent was to familiarize the reader as to why SSL was incorporated into Java via the Java Secure Sockets Extension (JSSE). After history, the current Java Security Architecture and the JSSE implementation of SSL were discussed. These two sections provided information and sources as to how SSL was implemented into Java. The chapter concludes with background information on certain attacks to which the Java implementation of SSL may be susceptible, as well as the tools used in this research.

3. Methodology

3.1 Introduction

This chapter focuses on the methodology and simulation setup of the research effort. It includes the problem definition, goals and hypothesis of the research. It also defines the system boundaries, system services, and performance metrics. From those definitions, the parameters and factors are identified. The chapter concludes with the experimental design, the evaluation technique, and the workload.

3.2 Problem Definition

The added flexibility and security provided by JSSE is attractive to commercial companies as well as government organizations. Before the use of the JSSE libraries becomes widespread within the DoD, it's prudent to perform an analysis on the code to find its strengths and weaknesses.

3.2.1 Goals and Hypothesis

The goals of this research are to determine the following:

- Does JAVA 1.4.2 use RSA blinding to prevent cryptographic timing attacks? It is believed that Java implemented the RSA blinding technique. This research determines if this is true and if it is the default configuration.
- Compare similar implementations of JSSE's implementation of SSL and OpenSSL's implementation of SSL. Specifically, determine the point at which each system's resources become exhausted and become unusable to the

SSL users, and also determine what/if any performance differences in terms of maximum number of secure sockets are derived from using 128, 192, and 256 bit symmetric keys within the AES, DES3, and RC4 encryption algorithms. Between the three different algorithms, AES is expected to perform significantly better since it's one generation ahead of the other two algorithms, and was selected through open competition to become the new standard replacing DES [NIST01]. The increased size of the symmetric keys will likely impact performance, though to what degree is unknown. Even though 128-bit symmetric key encryption is widely acknowledged as being virtually impossible to break through brute force, 192 or 256 would be better as long as there is not a significant reduction in speed.

3.2.2 Goal 1 Approach (Timing Attack Protection)

The second goal involves a well-known cryptographic vulnerability: timing attacks. Timing attacks enable an attacker to extract secrets from a secure system by observing the amount of time taken by the system to respond to various requests. Until recently, these attacks were thought to only apply to slower hardware systems such as smartcards. Attacks on web servers were unlikely since the decryption time of the encrypted text are masked by the many concurrent processes running on the system, as well as the latency introduced by propagation delay of networks in the client-server environment. Research at Stanford University has demonstrated these assumptions were false [BoB03]. The research conclusively showed that in an OpenSSL application it was indeed possible to implement a timing attack across a network against a RSA keying system [BoB03]. They

suggest a RSA blinding scheme to mask the calculation time required for decryption. During the research, it was determined with the release of Java J2SDK 1.4.2 RSA blinding is automatically implemented with the built in JSSE libraries. No further work was accomplished on this portion of the research.

3.2.3 Goal 2 Approach (Performance Comparison)

As a final goal, a performance comparison between four popular symmetric encryption algorithms provided in Java's implementation of SSL is conducted. Specifically, the point at which each system's resources are exhausted and the system is unusable to the SSL users is determined. Symmetric key size, JVM heap size, and symmetric encryption algorithm are all factors for this analysis.

3.3 System Boundaries

The system under test (SUT) is shown in Figure 4 and includes the SSL over TCP protocol system and the operating system network stack. The SUT does not include the workstations or servers, nor does it include any of the routers or physical wiring.

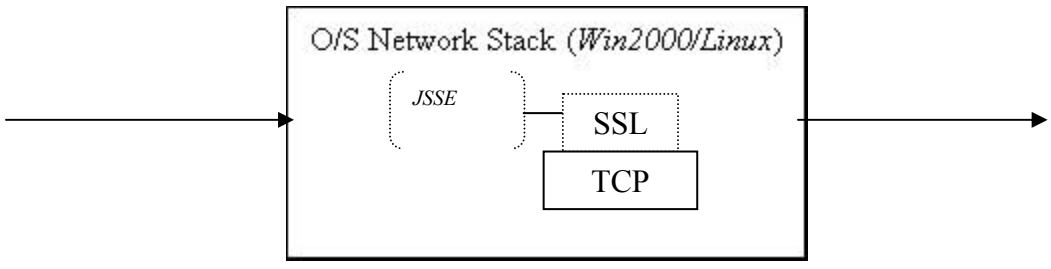

Figure 4: System Under Test

The components under test (CUT) are shown in italics in Figure 4. It includes the operating system and the JSSE implementation of SSL. Although there are other

implementations of SSL, this effort is limited to JSSE. Comparisons drawn between Win2000 and Redhat Linux should be based on a JSSE implementation on both operating systems.

3.4 System Services

The service provided by the system is an encrypted channel where data is securely passed. Possible outcomes are:

1. A successful secure transmission of data – the data is transferred between the client and server successfully and encrypted.
2. An unsuccessful transmission of data – the data does not reach its destination.
3. An unsecure transmission of data – the data is transferred between the client and server, but it is sent unencrypted.

3.5 Performance Metrics

The output performance metrics of this research are tied to the second goal defined above. The only metric for Goal 2 is the maximum number of secure sockets created between the clients and the server. The possible outcome of an unsecure transmission of data is excluded as a metric because a properly verified and validated SSL simulation does not send data in the clear.

3.6 Parameters

3.6.1 System

The system parameters include:

- JSSE version 1.0.3 is the standard packaged with Java SDK version 1.4.2 SE.
- The Cipher Block Chaining encryption mode of the symmetric keys is used.
- Block encryption requires padding to fill out the remainder of an unused packet. There is an option to not use padding, but it's not secure since it reveals the length of the data being sent, and within the confines of SunJSSE provider, it would preclude using symmetric encryption at all [Sun03].
- Symmetric key size is used because it directly impacts the performance and security of the data transmission.
- Hardware: 1GHz Intel Pentium III, 512 MB RAM, 20 GB HDD, 100 Mbit Fast Ethernet, Linux host on NIS domain with 1 server, Win2000 host on Win2000 domain with 2 domain controllers.

3.6.2 Workload

Workload parameters are limited to the number of concurrent SSL connections and the size of the data transmission. The concurrent SSL connections are used to establish when the SSL server is unable to service any more connections. The size of the data transmission is fixed so other factors can be studied.

The workload is applied within the logic of the simulation program. Each SSL connection is created as individual threads, and pauses for one second after every successful connection (see Figure 5). The one second pause is used to prevent premature overloading of the server. Also associated with the workload is the number of SSL sockets each client creates. A maximum number of 600 were implemented for each instantiation of the client software. The number 600 was chosen so as not to reach the

~734 socket maximum identified earlier in the pilot studies for a default heap size. The default heap size on the client JVM was not changed as part of the experiment. As the clients hit their maximum socket limit, more clients were initialized in order to reach the server maximum SSL socket limit. All the initialized sockets continued to pass traffic after initialization until the server reached it maximum limit.

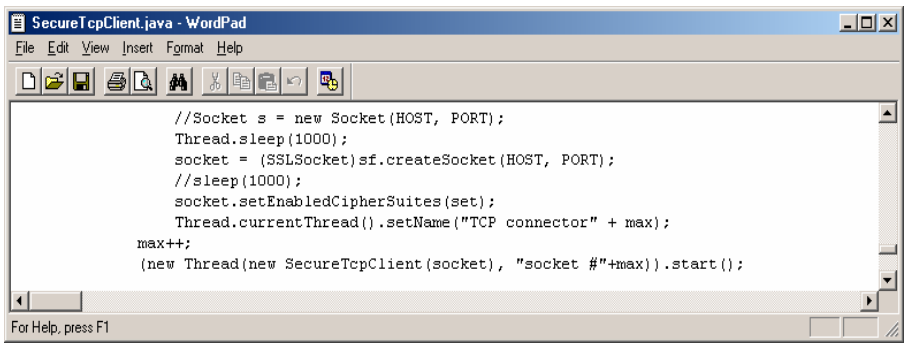

Figure 5: Multiple thread code

The size of the data transmission simulates three different types of SSL applications. The first data size of 16 bytes simulates an application with small data transactions. This is the smallest data size available for SSL data using cipher block chaining. Cipher block chaining effectively expanded any data block to the next 16 byte value. The second data size of 1418 bytes simulates large data transactions. Pilot studies showed that a data size of 1418 bytes was needed to completely fill the 1500 bytes of an Ethernet Frame. The third data size was chosen to split the difference between the first two data sizes and the data size of 768 was arbitrarily chosen.

3.7 Factors

Factors, their justification and values are listed below:

- Symmetric Key Size: 128, 168, and 256-bit. The key lengths are chosen based on the encryption algorithm and on analysis by an ad hoc group of cryptographers and computer scientists [BlD96]. According to the analysis, a 90-bit key is the minimum needed today to ensure the security of a symmetric key system against a brute force attack. Although this analysis is seven years old, it has held true over the past few years. For instance, there have been some successful brute force attacks on 128-bit symmetric key encryption, but the attacks have taken months to perform. The 128-bit key is the unofficial standard today, 168 bit is the only size allowed in DES3, and 256 is the largest key size currently available in Java 1.4.2 with the AES algorithm. The 128-bit keys are expected to be faster.
- Encryption Algorithm: RC4, AES, and DES3. These algorithms are some of the most commonly used today. AES lends itself to comparisons since it is the only one currently available in different key sizes within the Java 1.4.3 code. AES is expected to be a faster algorithm.
- Operating System: Windows 2000 and RedHat Linux 7.3 were chosen so some comparison can be drawn between two different SSL implementations. The JSSE version of Windows 2000 and Linux is stable and is used for analysis on both systems. The Linux implementation of JSSE will likely be faster than the Win2000 implementation of JSSE.

3.8 Evaluation Technique

As with the performance metrics, the evaluation techniques only apply to the fourth goal. The other three goals were met by verification of certain design methodology (range checking, defaults of Sun's PRNG, and the implementation of RSA blinding in the latest JSSE libraries). A prototype JSSE program was developed to determine the maximum socket limit when implementing SSL. The simulation program is validated by developing the program in accordance with a highly secure construct available within a JSSE program. A less secure design could enable more sockets to be established before server failure by not taxing the processor or memory with long encryption key calculations. The design addressed the hash function, the asymmetric encryption algorithm, and symmetric key encryption. The next several sections layout the design and how these functions were implemented to provide a secure simulation.

3.8.1 Cryptographic Hash Function

A cryptographic hash function is similar to a checksum. Data is processed with an algorithm that produces a relatively small string of bits called a hash. A cryptographic hash function has three primary characteristics: it is a one-way function, meaning that it is not possible to produce the original data from the hash; a small change in the original data produces a large change in the resulting hash; and it does not require a cryptographic key [Sun03].

When applying a hash to the message authentication code, it is sometimes referred to as a HMAC. HMAC can be used with any cryptographic hash function, such as Message Digest 5 (MD5) and Secure Hash Algorithm (SHA), in combination with a secret shared

key. HMAC is specified in RFC 2104. These also happen to be the two hash functions considered in the development of the simulation. SHA-1 was chosen because of it's slightly stronger encryption due to it longer hash value (160 versus 128 bit for MD5). No known attacks against SHA were found, while there have been a few against MD5 that could not be extended (to show inherent weakness in the algorithm). There is a slight tradeoff in speed. More information about MD5 and SHA can be found in the RFCs (1321 for MD5 and 3174 for SHA)

3.8.2 Asymmetric Encryption

The asymmetric encryption setup was more problematic in its implementation than other security functions. The implementation involved server side only authentication (ie., an https type application) with 1024 bit RSA Public Key encryption. The public key is wrapped in an X.509 certificate and is self signed (no certificate authority was used). Typically you would not use a self-signed certificate unless you trusted the source. For the simulation, a trusted third party was not necessary. After the server's public key was wrapped in a certificate, it was physically copied to each client used in the simulation. The server and client simulation programs were then written to incorporate a keystore and a truststore. The instructions for configuration and Java keytool usage is laid out in the JSSE Reference Guide [Sun03]. Figure 6 shows the source code used in the simulation program to implement the asymmetric key encryption design.

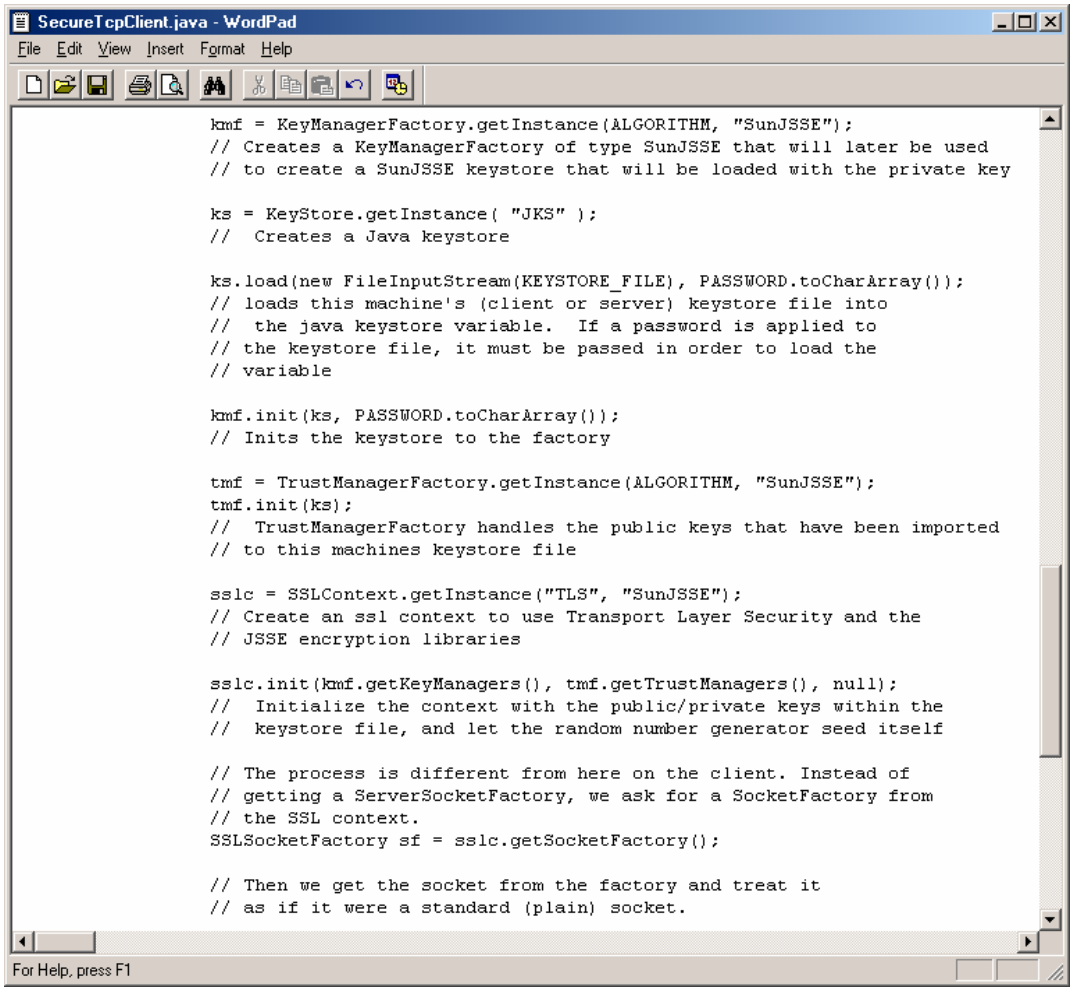

Figure 6: Source code for asymmetric key encryption.

3.8.3 Symmetric Key Encryption

The symmetric key encryption algorithms were chosen based on the factors of this analysis. JSSE has a default list of cipher suites it implements, and chooses them based on Table 1.

Table 1: SunJSSE supported cipher suites [Sun03].

Supported Cipher Suites in Default Preference Order		
Name	Enabled by Default	New in J2SE 1.4.2
SSL_RSA_WITH_RC4_128_MD5	X	
SSL_RSA_WITH_RC4_128_SHA	X	
TLS_RSA_WITH_AES_128_CBC_SHA	X	X
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	X	X
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	X	X
SSL_RSA_WITH_3DES_EDE_CBC_SHA	X	
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	X	X
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	X	
SSL_RSA_WITH_DES_CBC_SHA	X	
SSL_DHE_RSA_WITH_DES_CBC_SHA	X	X
SSL_DHE_DSS_WITH_DES_CBC_SHA	X	
SSL_RSA_EXPORT_WITH_RC4_40_MD5	X	
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	X	X
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	X	X
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	X	
TLS_RSA_WITH_AES_256_CBC_SHA *		X
TLS_DHE_RSA_WITH_AES_256_CBC_SHA *		X
TLS_DHE_DSS_WITH_AES_256_CBC_SHA *		X
SSL_RSA_WITH_NULL_MD5		
SSL_RSA_WITH_NULL_SHA		
SSL_DH_anon_WITH_RC4_128_MD5		
TLS_DH_anon_WITH_AES_128_CBC_SHA		X
TLS_DH_anon_WITH_AES_256_CBC_SHA *		X
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA		
SSL_DH_anon_WITH_DES_CBC_SHA		
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5		
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA		

Those items marked with an “*” require installation of the JCE Unlimited Strength Jurisdiction Policy Files.

This analysis only required four of the cipher suites, and one of these required the JCE Unlimited Strength Jurisdiction Policy Files. These can be found at http://java.sun.com/j2se/1.4.2/download.html. The four cipher suites used were:

- SSL_RSA_WITH_RC4_128_SHA (128 bit RC4 encryption with SHA hash)
- TLS_RSA_WITH_AES_128_CBC_SHA (128 bit AES encryption with SHA hash)
- SSL_RSA_WITH_3DES_EDE_CBC_SHA (168 bit 3DES encryption with SHA hash)
- TLS_RSA_WITH_AES_256_CBC_SHA (256 bit AES encryption with SHA hash)

Figure 7 shows the simulation program code implementing each of the cipher suites. Only one cipher suite was instantiated on the client for each simulation run. This ensured only the cipher suite being tested could be agreed upon by the server and the client.

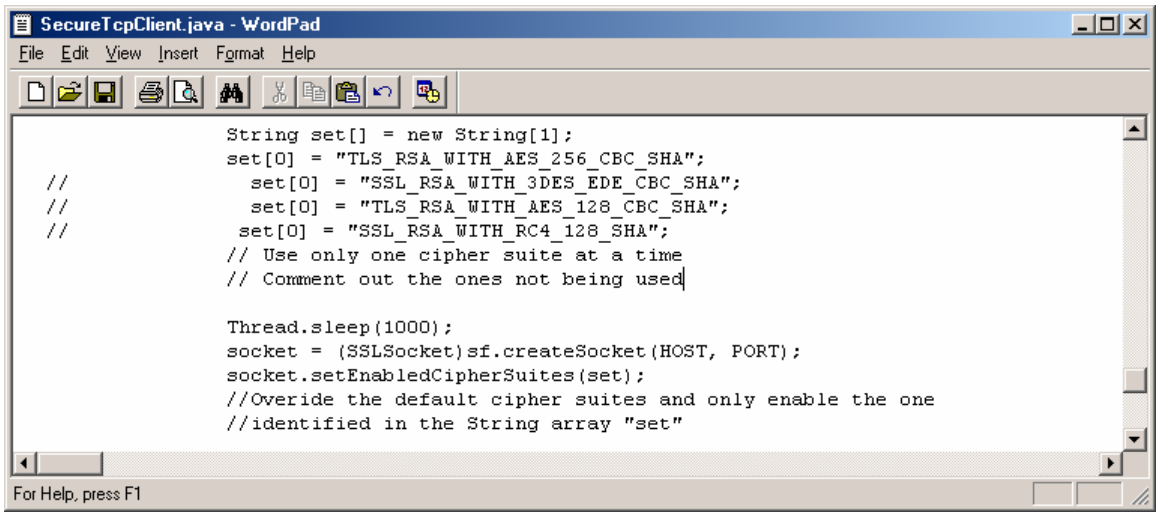

Figure 7: Symmetric key cipher suite implementation

3.9 Experimental Design

The experimental design is full factorial. It tests the failure/resource exhaustion of the JSSE implementation. The number of experiments required for a single sample value

of all the different combinations is: 4 (encryption algorithms) * 4 (heap sizes) * 3 (data sizes) = 48.

To determine the number of repetitions for the experiments, some assumptions are made about the maximum number of sockets. Based on pilot studies, the mean for a maximum number of secure sockets with the default heap size of 64 MB on the Java Virtual Machine (JVM) is 734.06. Observed variability for the pilot study data is 2.8625 sockets with a standard deviation of 1.69 sockets.

Testing for normality was used to determine if the Central Limit Theorem could be used to determine how many replications should be accomplished of each simulation. The method used here is the *Lilliefors test for normality*. The test basically compares the observed relative cumulative frequency distribution of the sample to that of the standard normal distribution. The two curves in the middle of Figure 8 represent the cumulative distribution of the observed data and a standard normal curve. Curves to either side represent the Lilliefors bounds for a sample size of 12 and a significance of 0.01 (99 percent confidence). If the observed relative cumulative frequency falls outside the bounds given for the specified sample size, then the data is not from a normal distribution. Since the curve representing the observed data does not fall outside the bounds, we can be 99 percent confident that the data is from a normal distribution. Therefore, the data is considered normally distributed and the Central Limit Theorem is used to analyze the data.

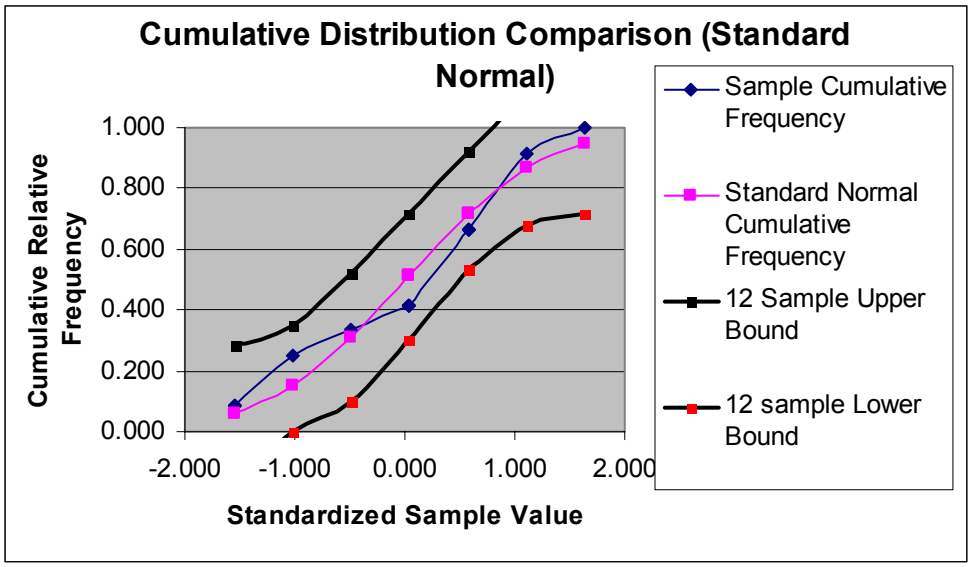

Figure 8: Pilot Study (Lilliefors graph)

Based on these observations, the following equations, and the assumption that normality will hold on larger heap and data sizes:

1. $\bar{x} \approx N(\mu, \sigma/\sqrt{n})$ (sample mean calculation).
2. $C.I. = \bar{x} \pm t^* \sigma/\sqrt{n}$ (calculation for a Confidence Interval).

Using previous assumptions about the mean and standard deviation, as well as a confidence interval of 90% and plugging those numbers into the equations:

$$C.I. = 734.06 \pm 2.353 * (2.8625/\sqrt{4}) = (730.692, 737.428).$$

Therefore, four repetitions of the 48 experiments are likely needed. A total of 192 experiments is performed.

3.10 Summary

The focus of this chapter is on the simulation design and methodology of the research effort. It includes the problem definition, as well as defined the system

boundaries, system services, and performance metrics. From those definitions the parameters and factors were identified and discussed. The chapter concludes with the experimental design, the evaluation technique, and the workload used in the analysis effort.

4. Analysis and Results

4.1 Chapter Overview

This chapter discusses the analysis and results of the research effort. Simulation data is presented and explained in order to answer the investigative questions of Chapter 3. The chapter concludes with a summary of the findings.

4.2 Results of Simulation Scenarios

A quick scan of the data reveals that, as the heap size increases, so does the number of secure sockets that can be created (see Table 2). This is because each socket requires a fixed amount of resources from the heap. Creating sockets without closing any eventually uses up all the space allocated in the heap. The first section of Table 2 shows the average number of sockets each encryption algorithm supports with the associated heap size. There are twelve values averaged for each cell, three data sizes (16, 768, and 1418) and four replications of each unique configuration. The second section of Table 2 shows the standard deviation for the values that were averaged in the first part.

Appendix A shows the data gathered for the socket limit. The naming convention is as follows: AES128h64d1418_win indicates the AES 128-bit encryption algorithm, with a JVM heap size of 64 MB, an encrypted data size of 1418 bytes, and the simulation server was run on a Win2000 operating system.

Table 2: Average/Standard Deviation of Maximum Number of Secure Sockets

		Encryption Algorithm			
	Average	RC4	AES128	AES256	DES3
Heap size	64 MB	736	734	729	744
	96 MB	1106	1104	1095	986
	256 MB	2523	2289	2211	1673
	384 MB	2589	2488	2412	1769

	Standard Deviation	RC4	AES128	AES256	DES3
Heap size	64 MB	2	2	1	1
	96 MB	3	3	2	199
	256 MB	490	575	603	974
	384 MB	720	869	904	1343

Another interesting result is the different maximum numbers of sockets between the encryption algorithms given a heap size of 256 and greater. The standard deviation is a measure of variability in the data. A rise in the standard deviation is a sign of increased variability in the data samples. Table 2 provides some insight into different maximum number of sockets occurring in the different algorithms. For DES3 the data variability increases with a heap size of 96 MB. The other algorithms increased variability does not begin at this point. The DES3 simulations are used to explain what is happening.

Table 3 shows the average maximum socket values recorded for the DES3 simulations with heap sizes of 64 MB, 96 MB, 256 MB, and 384 MB. As evident from Table 3, there is not much variance between the samples in the 64 MB heap size. However, a heap size of 96MB shows a significant drop (35 percent) in the number of secure sockets that can be created with a data size of 1418 bytes. This shows that data

size is probably going to be a significant factor in the variance of the data. This assertion is proved later on in this chapter when the variance analysis is conducted.

Table 3: Averages for DES3 Maximum Number of Secure Sockets

		Encrypted Data Size (bytes)		
	Average	1418	768	16
Heap Size	64MB	743	743	745
	96 MB	721	1117	1119
	256 MB	936	1095	2988
	384 MB	723	1003	3580

At this point, it appears something else besides heap size is limiting the maximum number of secure socket creation. It is interesting that the 256 MB heap size showed improvement while the 384 MB heap size did not. Taking this into account, we need to see if the mean is a good statistic to use for comparing the maximum number of sockets. For example, it is not very useful to use the mean when two sample values between replications are significantly different. The standard deviation of the data, given in Table 4, can help determine if this is the case.

Table 4: Standard Deviation for DES3 Maximum Secure Sockets

		Encrypted Data Size (bytes)		
	Standard Deviation	1418	768	16
Heap Size	64MB	1	1	1
	96 MB	71	2	2
	256 MB	42	34	18
	384 MB	37	49	16

It is evident there is much more variability in the data for heap sizes of 256 and 384, and also for a heap size of 96 MB with a data size of 1418 bytes. Since this is the case, the mean for those instances just show that generally speaking, there is a decrease in the maximum number of sockets as the data size gets larger. The high variance in the data at the 256 MB and 384 MB levels still falls within acceptable limits for the four replication simulation. The next section evaluates what resource could be limiting the maximum socket number, and provides answers as to why it's impacting the variance of the simulation data more so than the smaller heap sizes did.

A computer system has a finite set of resources. The limiting resource could be a number of things (main memory, virtual memory, CPU, and network bandwidth). By limiting the heap size to a fraction (75 percent) of available main memory, it was made a factor for variation and analysis. It was also a resource capable of being monitored for resource exhaustion. Given the controlled environment, network bandwidth and CPU utilization are the only other resources that could account for the high data variance across simulations. Network bandwidth is a limiting factor if the socket creation fails due to waiting on network bandwidth where the client request times out before receiving an answer from the server. In order to rule out network bandwidth, the round trip time for DES3 secure socket connections was examined. Since a heap size of 96 MB had significant variation in the maximum number of sockets created, this data is examined to see if network bandwidth had an impact. If network bandwidth was responsible for the inconsistencies, there should be a trend upwards over time. Figures 9, 10, and 11 show the round trip time for the DES3 samples with a heap size of 96 MB.

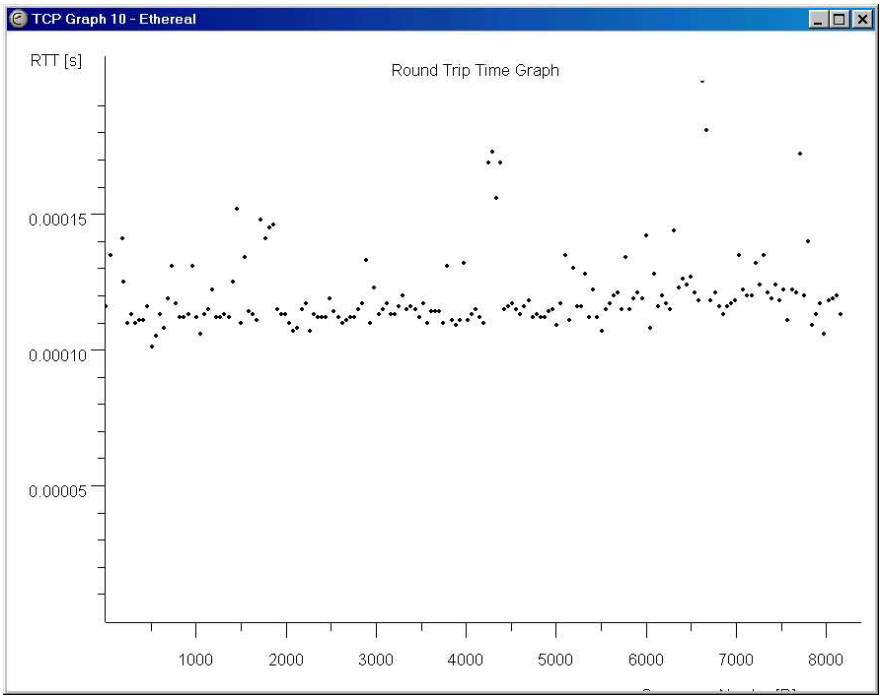

Figure 9: Round Trip Time (RTT) for 16 Byte Data Packets, 96 MB Heap, and DES3 Encryption.

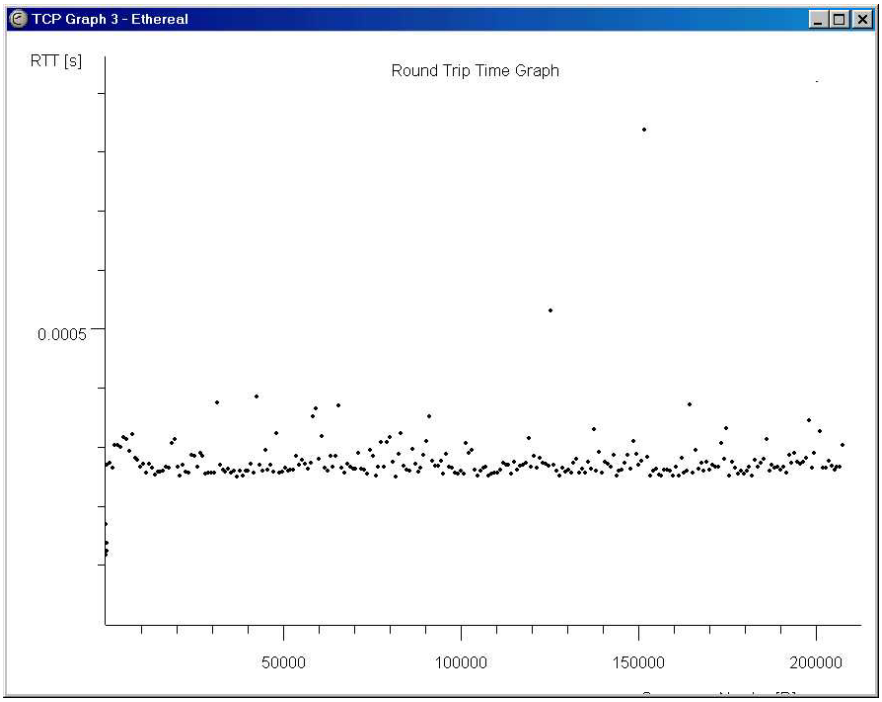

Figure 10: Round Trip Time (RTT) for 768 Byte Data Packets, 96 MB Heap, and DES3 Encryption

The y-axis on these figures represent the round trip time in seconds for one data packet. The x-axis represents the sequence number of the Ethernet packets. If bandwidth was causing the bottleneck, some type of rise along the y-axis indicating an increase in the RTT would be observable. This is not occurring, and therefore network bandwidth can be ruled out as the limiting resource. In the 16-byte and 768-byte simulations, the resource limit was on the heap size. This was demonstrated by the "out of memory" error received while running the simulation.

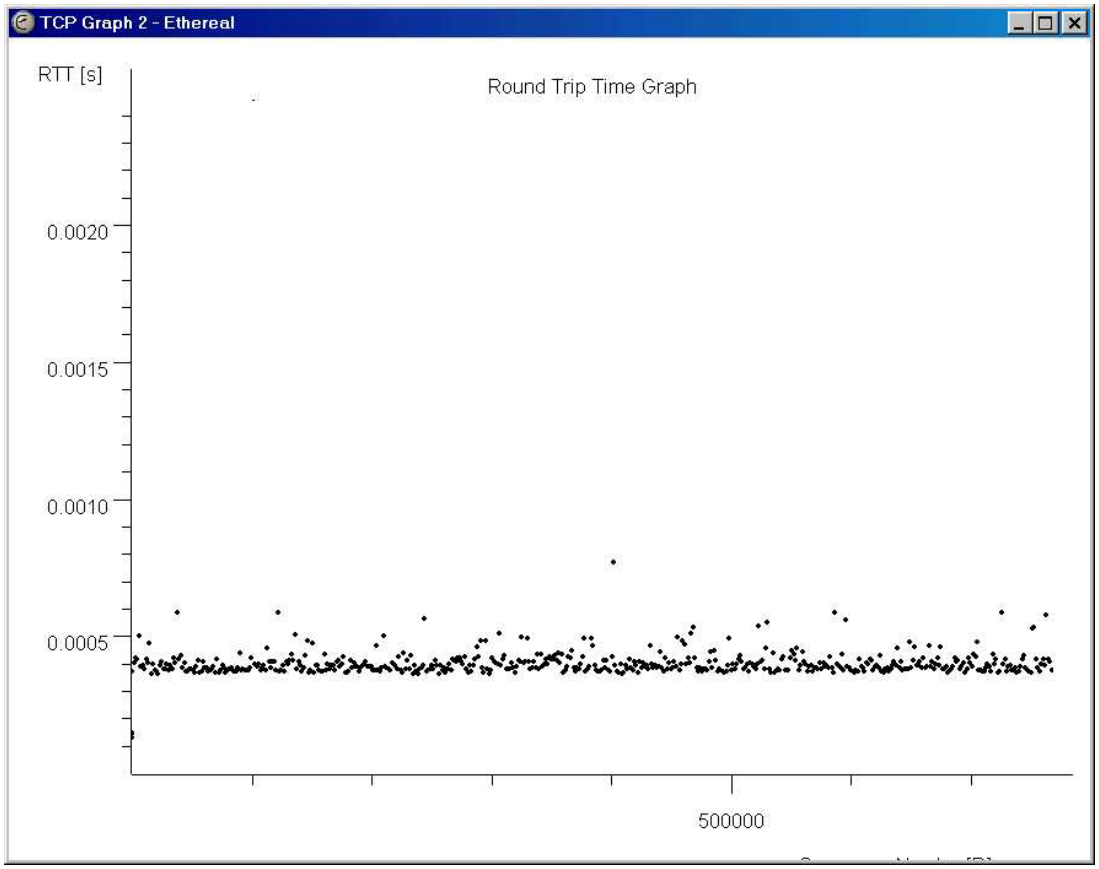

Figure 11: Round Trip Time (RTT) for 1418 Byte Data Packets, 96 MB Heap, and DES3 Encryption

For the 1418 byte data size neither the memory nor the network bandwidth is the limiting resource because there was no "out of memory" error, and Figures 9, 10, and 11

show network bandwidth was not the issue. Therefore, the CPU utilization must be considered as a limiting performance factor. Using the same logic as in the network bandwidth analysis, the analysis once again used the DES3 encryption algorithm with a heap size of 96 MB. If CPU utilization is the limiting resource, its graph over time should show an incline up towards 100 percent utilization. This analysis examines the CPU utilization for the Java process. Note that variability in the utilization can occur as the process waits for CPU time. Figures 12, 13, and 14 illustrate that data sizes of 768 and 1418 bytes have a much greater impact on the CPU utilization than a data size of 16 bytes.

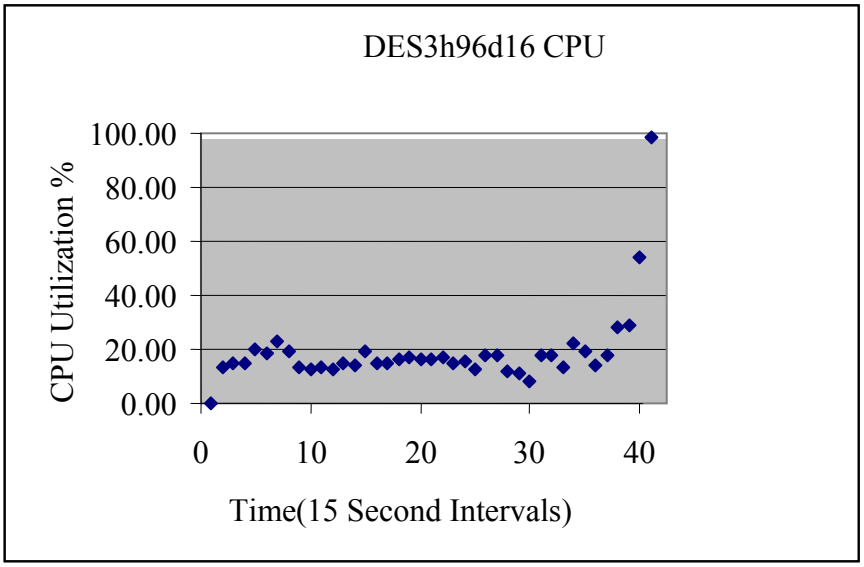

Figure 12: CPU Utilization for DES3 Simulation With Heap Size 96MB and Encrypted Data Size of 16 Bytes

When the memory resource (heap size) is used up, the program terminates giving an "out of memory" error. Since each socket requires the same amount of resources, the difference between replications was very small. However, when CPU was at 100 percent utilization, the program behaved more erratically. Socket creations would sometimes fail

and sometimes succeed. Simulations were terminated when five consecutive secure connections were unable to be created and pass encrypted data. It was difficult to predict whether this would occur or not when the CPU was fully utilized.

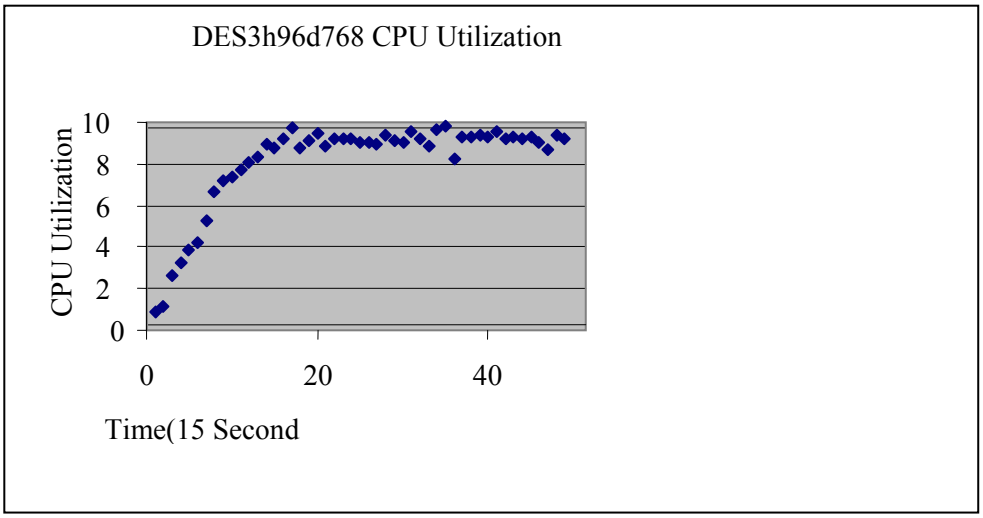

Figure 13: CPU Utilization for DES3 Simulation With Heap Size 96MB and Encrypted Data Size of 768 Bytes

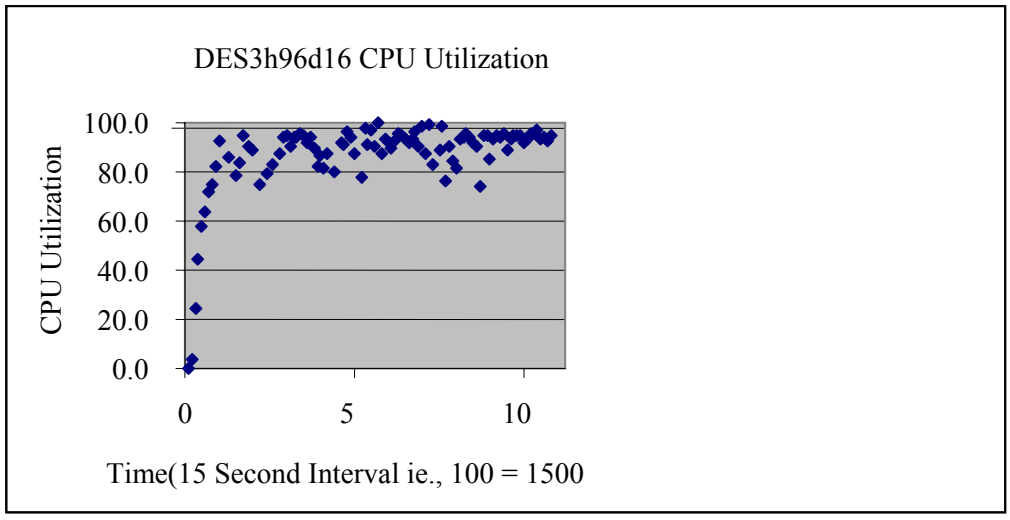

Figure 14: CPU Utilization for DES3 Simulation With Heap Size 96MB and Encrypted Data Size of 1418 Bytes

The data shown in Appendix B illustrates the relationship between the number of sockets created and the associated CPU percentage for all simulations performed with a heap size of 384 MB. From these graphs, while all the algorithms hit 80 percent CPU utilization much faster when the encrypted data size is large (1418 bytes), DES3 and AES256 appear to be more affected, with DES3 being the most impacted. This indicates the higher the symmetric key size, the more load on the CPU. The CPU utilization presented here is only what is used by the Java process. It is not the CPU utilization of every process on the server. The data shown in Appendix B also indicates the 168-bit DES3 algorithm is not as efficient as the 256-bit AES256 algorithm. This is based on how quickly the DES3 algorithm reached 80-100 percent CPU utilization. This validates a reason why the AES algorithm was chosen as the new standard to replace DES3. RC4 and AES128 were much more efficient as far as CPU utilization, which is expected since these algorithms only use a 128-bit symmetric key. Still to be determined is what has the most impact on the number of secure sockets that can be created between all the different factors (encryption algorithm - A, heap size - H, and encrypted data size - D). This question is answered by performing an analysis of variance (ANOVA).

Using the data in Appendix A, an ANOVA table is created to easily determine what factors have statistically significant impact on the maximum number of sockets that can be created. There are three main effects, three first-order interactions, and one second-order interaction. Table 5 shows the ANOVA table used to perform the analysis.

There were a total of 192 experiments used to analyze the maximum number of secure sockets created. Appendix A lists the measured values from the experiments. As

is seen, the number of sockets has a wide range, the ratio of which is 3300/698, or about 5. Performing a log transformation of the data stabilizes the variance in the data. Table 6 shows the transformed data.

Table 5: ANOVA Table for Maximum Socket Analysis

Component	Sum of Squares	Percentage of Variation	Degrees of Freedom	Mean Square	F-Computed	F-Table	Significant at 0.05 level
SSY	1891.09		192				
SS0	1869.76		1				
SST	21.33	100	191				
Main Effects							
SSD	1.12	5.23	2	0.56	57.24	3.1	Significant
SSH	6.83	32.04	3	2.28	233.68	2.72	Significant
SSA	0.53	2.51	3	0.18	18.28	2.72	Significant
First Order Interactions							
SSDH	8.32	39.02	6	1.39	142.31	2.2	Significant
SSDA	0.01	0.07	6	0.002	0.24	2.2	Not Significant
SSHA	0.41	1.91	9	0.05	4.65	2	Significant
Second Order Interactions							
SSDHA	2.7	12.64	18	0.15	15.37	1.72	Significant
Errors							
SSE	1.40	6.58	144	0.01	1		

Table 6: Log Transformed Socket Data

		Heap Size			
Algorithm	**Data Size**	64	96	256	384
AES128	16	2.87	3.04	3.46	3.56
	768	2.87	3.04	3.38	3.33
	1418	2.87	3.04	3.20	3.22
AES256	16	2.86	3.04	3.47	3.56
	768	2.86	3.04	3.34	3.31
	1418	2.86	3.04	3.18	3.20
DES3	16	2.87	3.05	3.48	3.55
	768	2.87	3.05	3.04	3.00
	1418	2.87	2.86	2.97	2.86
RC4	16	2.86	3.04	3.47	3.55
	768	2.86	3.04	3.44	3.37
	1418	2.87	3.04	3.27	3.28

From Table 5, all the factors except the first order interaction between data size (D) and encryption algorithm (A) were statistically significant. However, there were a couple that were much more significant. Data size, heap size, and the first order interaction between data and heap size had the most impact on the variance (over 75 percent between those three).

From this analysis, it can be concluded that data size and heap size have the most impact on the variance of the number of secure sockets that can be created given the test system. However, it is of primary interest to find the best combination of the three factors for providing efficient means to create the greatest number of secure sockets without reaching a system resource limitation. The range method was used to find the average response corresponding to each level of the factor, and then to find the difference between the maximum and the minimum of the averages. A factor with a large range is

considered important. Table 7 shows the data for this method. Table 7 columns indicate the level of the factor. Data Size level 1 is 16 bytes, level 2 is 768 bytes, and level 3 is 1418 bytes. Algorithm level 1 is AES 128-bit, level 2 is AES 256-bit, level 3 is DES3 168-bit, and level 4 is RC4 128-bit. Heap Size level 1 is 64 MB, level 2 is 96 MB, level 3 is 256 MB, and level 4 is 384 MB.

Table 7: Factor Averages and Range for Secure Sockets

	16, AES128, H64	768, AES256, H96	1418, DES3, H256	RC4, H384	Range of Averages
Data Size	2091	1418	1170	none	921
Algorithm	**1654**	**1612**	1293	**1737**	444
Heap Size	735	1073	**2174**	2314	1579

From Table 7, the items in bold are considered the best in terms of ranges. Data sizes are excluded from consideration since these values should not be limited based on the external performance limitations of the secure system. With a value of 1293 average sockets created, DES3 was the worse performing algorithm in the study. AES128, AES256, and RC4 had similar performance statistics. With the additional security provided by the longer key size in AES256, it is more attractive at this point than either AES128 or RC4. For a heap size setting, it is believed that a heap size of 256 MB is better in this configuration, because it offered the greatest increase in performance. It achieves an average of a 200 percent increase over a heap size of 64 MB, a 100 percent increase over a heap size of 96 MB, and just a 6 percent decrease over the average that was achieved with a 384 MB heap size

4.3 Investigative Questions Answered

This section documents the answers for the investigative questions posed in Chapter 3 of this document. The questions and corresponding answers are documented in the next several sections of this thesis.

4.3.1 RSA Blinding

Does JAVA 1.4.2 use RSA blinding to prevent cryptographic timing attacks? It was believed that Java implemented the RSA blinding technique. This research determined that with the release of Java 1.4.2, it is indeed implemented and is the default. In fact, no way was found to turn it off and still use RSA as the Public Key Encryption algorithm.

4.3.2 Resource Exhaustion

Another research question was to determine the point at which some resource on the system is exhausted and the system becomes unusable to SSL users. This area also determines what, if any, performance differences in terms of maximum number of secure sockets are derived from using 128, 192, and 256 bit symmetric keys within the AES, DES3, and RC4 encryption algorithms. Between the three different algorithms, AES was expected to perform significantly better. This is due to the fact that it is one generation ahead of the other two algorithms, and was selected through open competition to become the new standard replacing DES [NIST01]. The increased size of the symmetric keys did impact performance, though with AES, not to a great degree with respect to maximum number of secure sockets (Table 7). Even though 128-bit symmetric key encryption is widely acknowledged as being virtually impossible to break through brute force, 192 or

256 is better if there is not a significant reduction in speed. This analysis implies that the best setup, with respect to getting the most secure sockets would be to use a heap size of 256 MB, and the AES 256-bit symmetric key encryption.

4.4 Summary

This chapter discussed the analysis and results of the research effort. The simulation data was presented. This data revealed the best possible setup of the given factors with regards to getting the maximum number of secure socket connections while still maintaining the highest level of security. The data was explained to answer the investigative questions posed in Chapter 3.

5. Conclusions and Recommendations

5.1 Research Overview

The Java SSL/TLS package distributed with the J2SE 1.4.2 runtime is a Java implementation of the SSLv3 and TLSv1 protocols. Java-based web services and other systems deployed by the DoD will depend on this implementation to provide confidentiality, integrity, and authentication. Security and performance assessment of this implementation is critical given the proliferation of web services within DoD channels.

This chapter discusses the conclusions of the research and talks to the significance of the effort. The other sections of this chapter discuss the conclusions reached from the study, the significance of the research effort, recommendations for actions, and recommendations for future research.

5.2 Conclusions of Research

The research outcome centers around what steps should be taken to build a highly secure JSSE program while still allowing a maximum number of secure connections. Given the hardware constraints of this effort, it was determined that a server-side only authentication mechanism with a 1024-bit RSA asymmetric key would be used as a parameter to find the best combination of symmetric key size, symmetric key encryption algorithm, and JVM heap size for building a high performance, highly secure system.

The best performance combination was a JVM heap size of 256 MB, the AES symmetric key encryption algorithm, and a symmetric key size of 256-bits. Comparable

results were determined for heap sizes of 256 MB and 384 MB. The main reason for staying with a 256 MB heap size was that it limits the impact on the physical memory of the server. There was minimal value added since there was only an 8 percent increase between the two heap sizes in the maximum number of sockets that could be created by a server using the 256-bit AES encryption algorithm.

5.3 Significance of Research

This research is significant because it determines what parameters and factors to consider when developing a high-use Java JSSE server. High-use is defined as being over 2000 simultaneous secure socket connections. It also documented how to set up the application securely and provides source code that could be used as the security engine for a secure client/server application.

5.4 Recommendations for Action

It is recommended that before any development begins, a thorough investigation of the latest security issues associated with the development libraries should be determined. There were several security issues fixed from the Java J2SE 1.4 release to the J2SE 1.4.2 release.

The latest version of the development libraries should be used to the fullest extent possible. Old applications written in previous releases should be reviewed to ensure they address all the latest vulnerabilities, and rewritten where necessary to utilize the security fixes implemented in the latest J2SE release.

Finally, based on the results of this effort, there is not a significant performance difference between 256-bit AES encryption and some of the more popular 128-bit symmetric encryption algorithms. The 256-bit AES encryption mechanism does represent a significant increase in the ability to thwart brute force attacks. It is suggested that organizations download the *JCE Unlimited Strength Jurisdiction Policy Files* and use the 256-bit AES algorithm for the secure transmission of data.

5.5 Recommendations for Future Research

This section is intended to capture all the correspondence and conversations with the National Security Agency (NSA) sponsor in defining exactly what this research was going to cover. The following list of research extensions were of interest to the sponsor during the conduct of this investigation. Before future work is attempted, coordination with the sponsor for elaboration of these topics is prudent.

5.5.1 SSL standards compliance

Topics:

- Correct handling of SSL messages and roles.
- Proper responses to invalid messages, including fail-secure behavior.
- Error behavior including leakage of information in error messages.

Tools and Techniques:

- A good packet analyzer, like Ethereal 0.9.11.
- Use a maliciously modified version of OpenSSL to do most of this testing.

5.5.2 SSL security configuration and lockdown

Topics:

- How are choices of cipher suites configured from code and in global configuration?
- Are cipher suite choices enforced correctly on the client and on the server?
- Server configuration of client authentication.
- Access by application code to security information.
- Degree to which application code can control SSL, including application code ability to configure SSL improperly.

Tools and Techniques:

- A good packet analyzer
- An SSL debugging proxy might be good here
- Mostly, this topic area will involve writing JSSE test code in Java, then sniffing traffic
- Instrumented version of OpenSSL might be good

This research effort covered a majority of this area. However, there could be more focus put on specific items that can be derived from this area. For instance, no testing was performed to create a misconfigured client that could possibly force a "secure" server to a more easily hacked symmetric key algorithm like DES 40-bit.

5.5.3 PKI Security

Topics:

- Certificate management and trusted roots; how is integrity of cert storage protected, administered, and trusted.

- Ability to perform cert chain validation; RFC2459 and PKIX compliance, recognition and enforcement of X.509v3 certificate extensions, and use of CRLs, resistance to certificate chain spoofing.
- Visibility of certificate operations to application code.
- Date and validity interval handling.
- Private key import capabilities and handling of PKCS#12 files.

Tools and Techniques:

- Mostly writing JSSE test code.
- A good ASN.1/BER decoder, like dumpasn1.
- A set of certificate chain test cases (e.g. from Santosh at U. of MD).
- Ability to generate certificates and CRLs (probably OpenSSL is sufficient for this.

5.5.4 JSSE/JCE randomization and key generation

Topics:

- Random Number Generation (RNG) operation and state exposure
- RNG seeding, including JCE v. JSSE
- RNG structure and reseeding

Tools:

- Some test code exercising JSSE crypto provider
- Java debuggers
- Java decompilers (This topic is likely to require some reverse-engineering of the Java code, because the source of the RNG is not available.)

5.5.5 Robustness of JSSE Implementation

Topics:

- Reaction to bad SSL packets.
- Reaction to bad certificates.
- Interoperability with other SSL implementations, especially at the 'edges' of the specification.
- Susceptibility to timing attacks and other active cryptographic attacks (e.g. Bleichenbacher attack).
- Susceptibility to resource exhaustion or other denial of service attacks.
- Presence of information leaks (e.g. in padding, nonces, etc.).

Tools:

- Hacked SSL implementation (maybe OpenSSL).
- Traffic generator (e.g. Hailstorm).
- X.509 test case generator (NSA C4 can provide a simple one).

This is another area that was partially covered by this thesis effort. Bad SSL packets generated at the client were tested, but man-in-the-middle attacks were never performed in the simulations. Timing attacks were addressed through research, not simulation and resource exhaustion limitations were found with given parameters and factors. Information leaks were lightly touched upon.

5.6 Summary

The background chapter set up the research effort by giving detailed background on Java and how it's evolved to become a secure architecture. The SSL and JSSE implementations of it were explained. The handshake and record protocol within SSL were mapped to show how the SSL process of encryption actually works. JSSE class mappings and SSL interaction figures were shown and explained, as were security concerns associated with any SSL implementation.

The methodology chapter stated the research problem and the goals of the research. This chapter also laid out the system and the services provided by the system. Performance metrics were defined and parameters and factors were identified. The workload and its implementation were explained as were the details of how the factors and parameters were implemented in the simulation program.

Chapter Four focused on the analysis and results of the experimentation. The raw data obtained from the simulation scenarios was presented. The analysis of the data is explained and the answers are matched to the investigative questions of the research effort.

This chapter discussed the conclusions of the research and talked to the significance of the effort. The other sections of this chapter discussed the conclusions reached from the study, the significance of the research effort, recommendations for actions, and recommendations for future research.

Appendix A

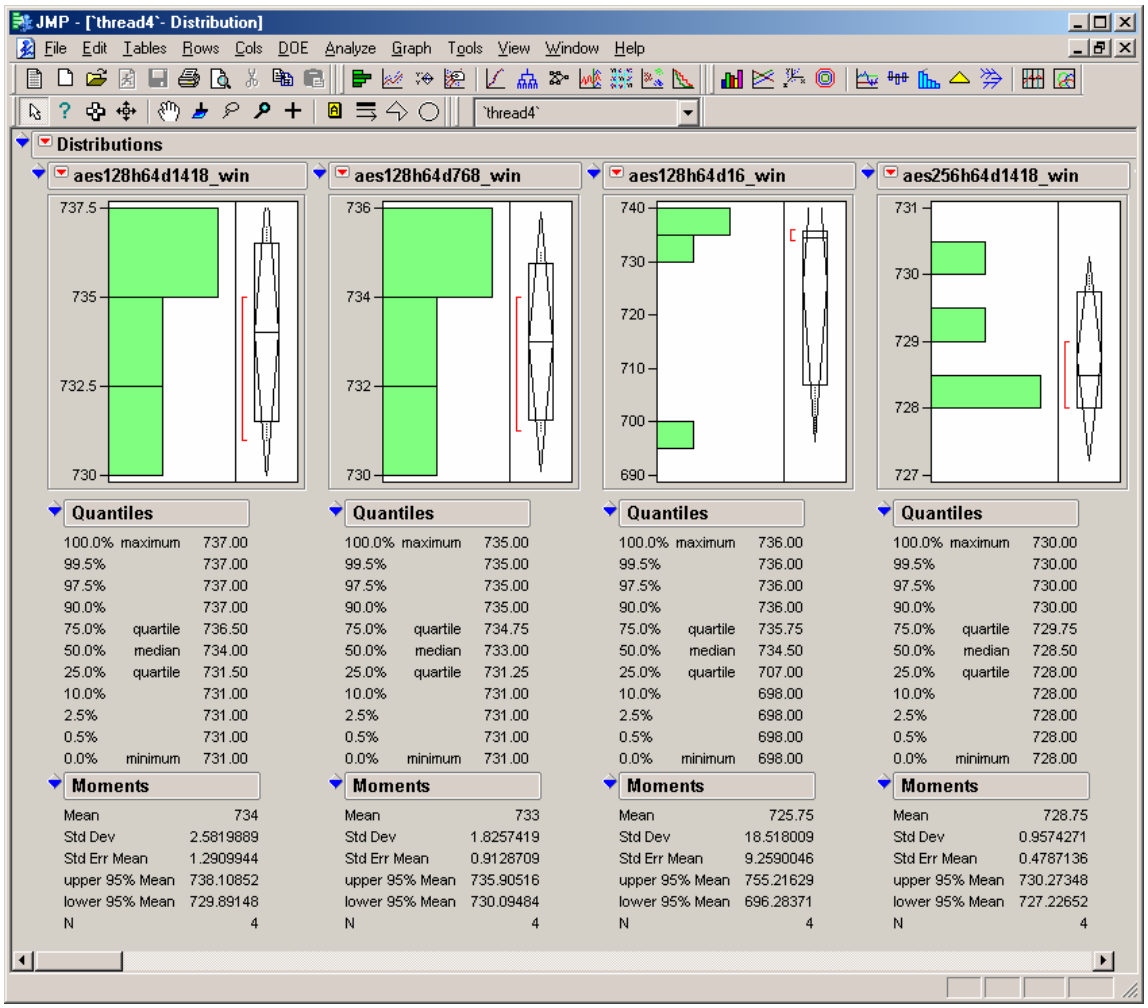

Figure A.1: JMP Distribution Data for Heap Size of 64 MB (Part 1)

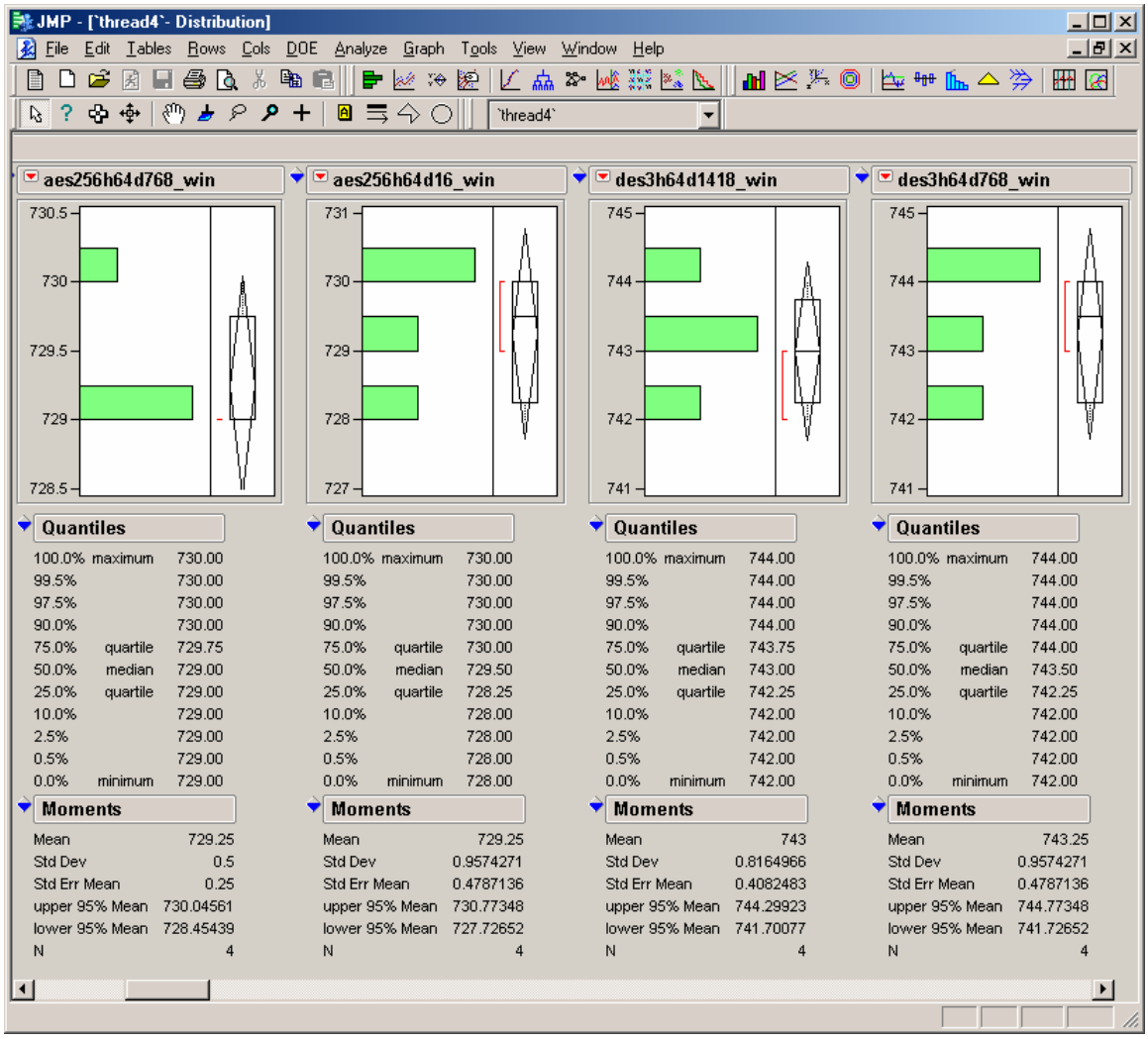

Figure A.2: JMP Distribution Data for Heap Size of 64 MB (Part 2)

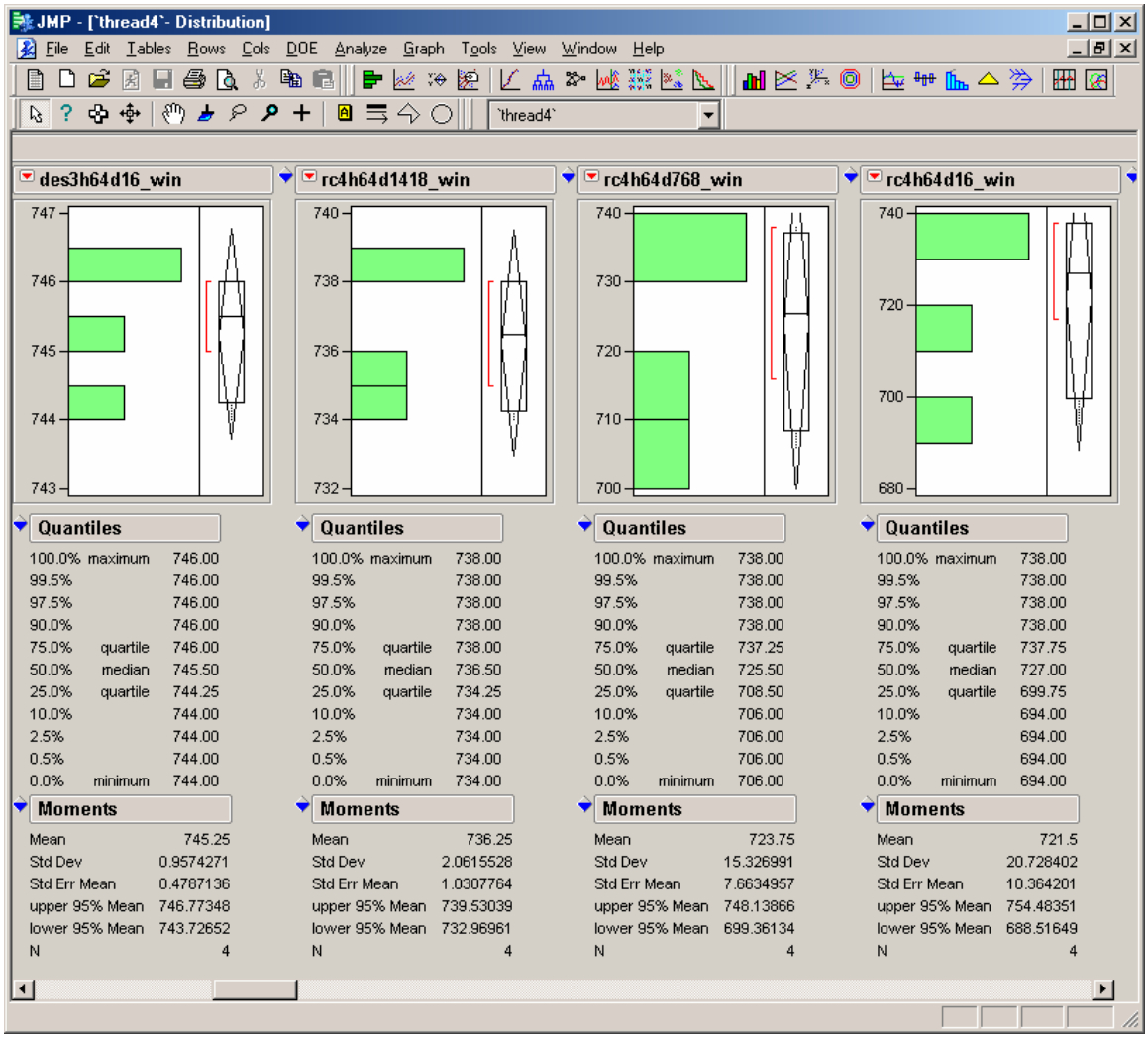

Figure A.3: JMP Distribution Data for Heap Size of 64 MB (Part 3)

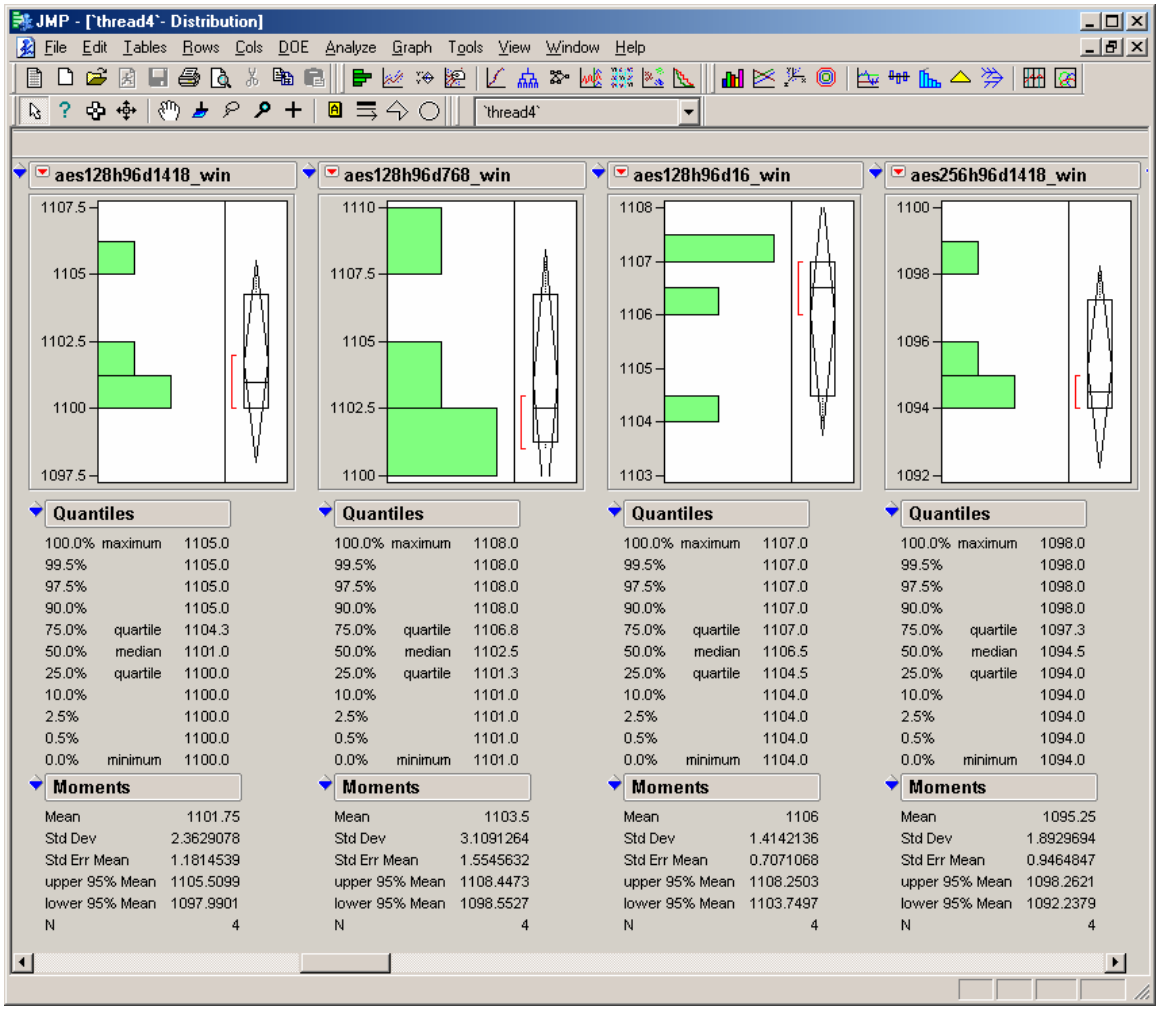

Figure A.4: JMP Distribution Data for Heap Size of 96 MB (Part 1)

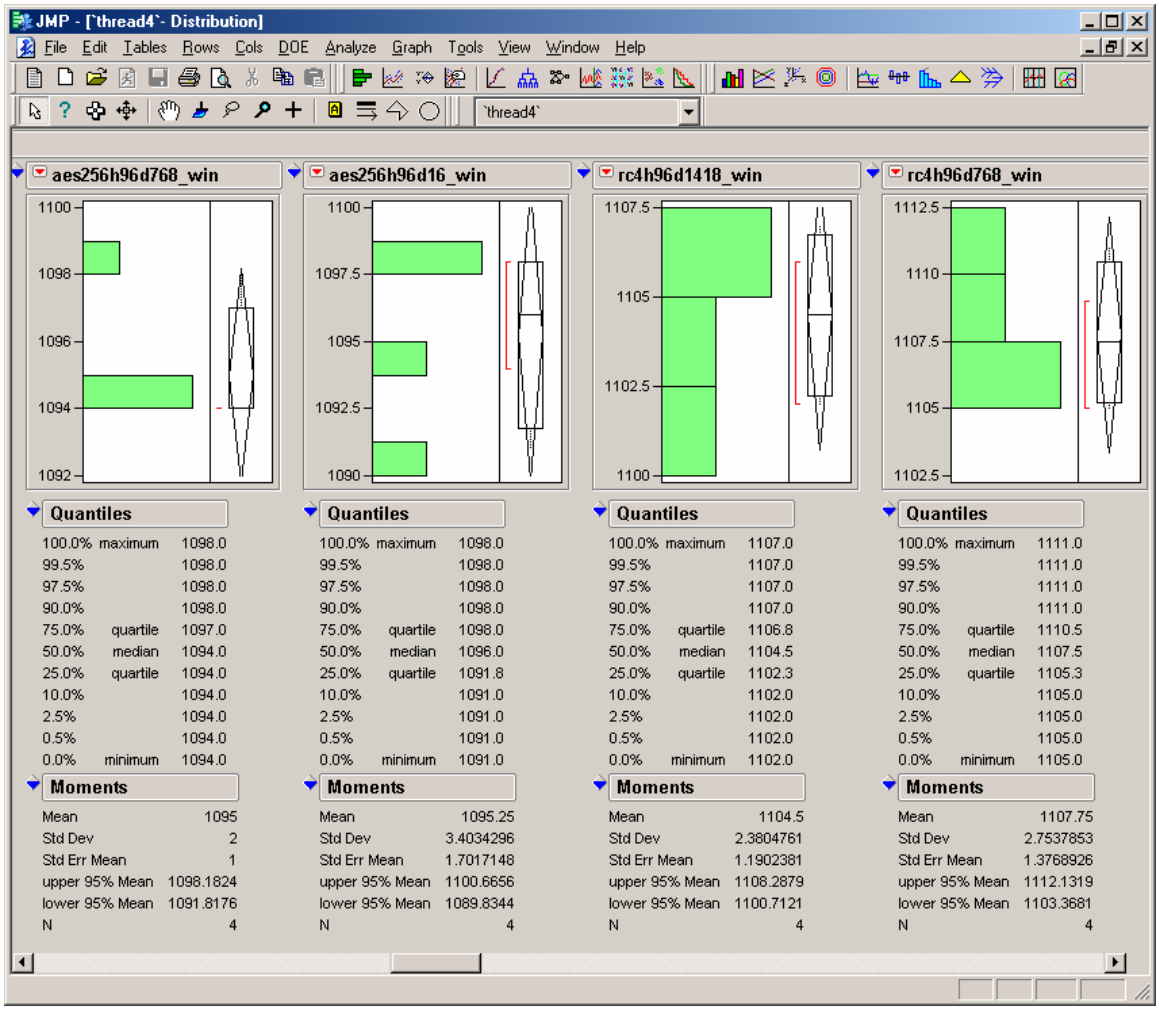

Figure A.5: JMP Distribution Data for Heap Size of 96 MB (Part 2)

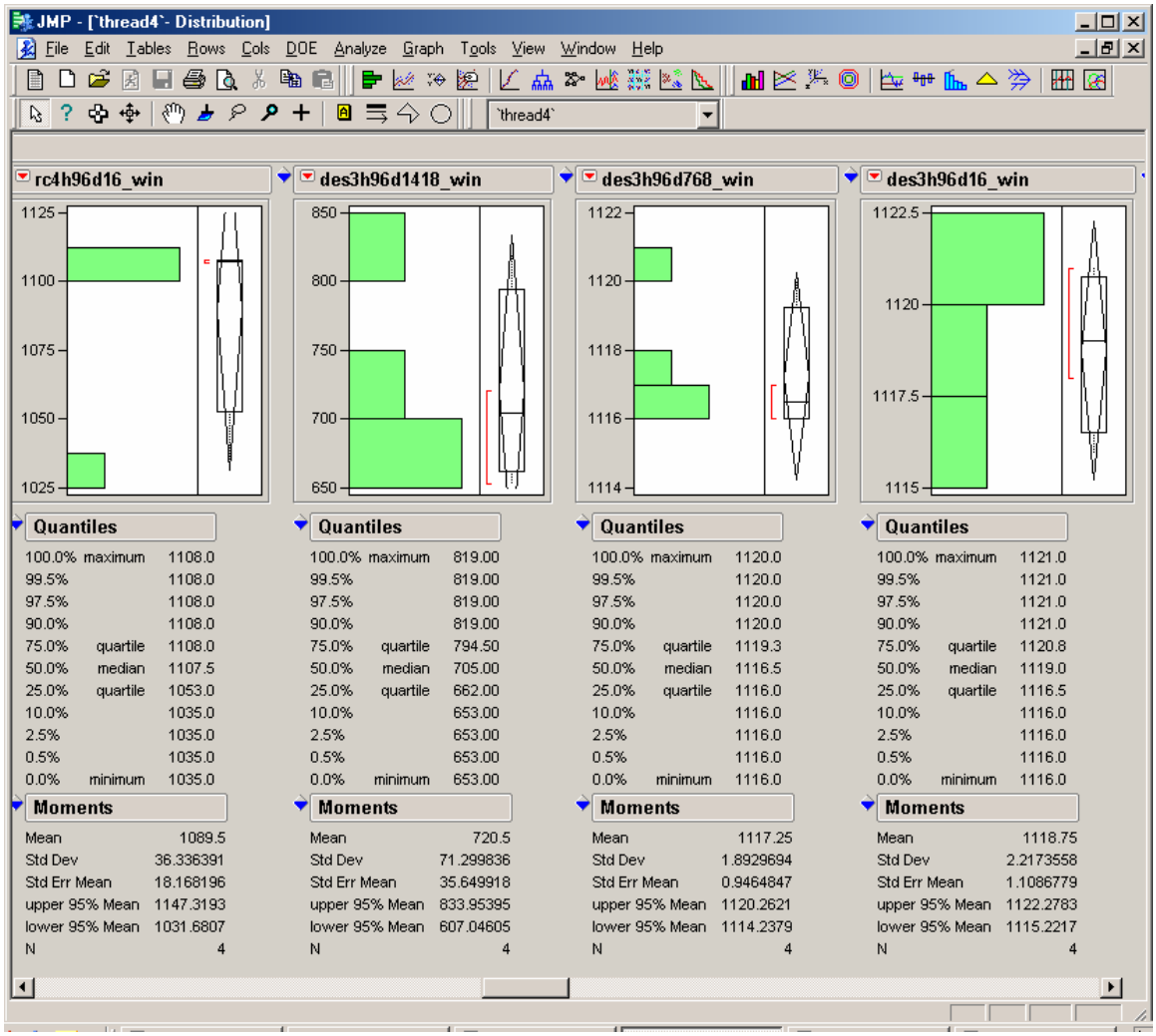

Figure A.6: JMP Distribution Data for Heap Size of 96 MB (Part 3)

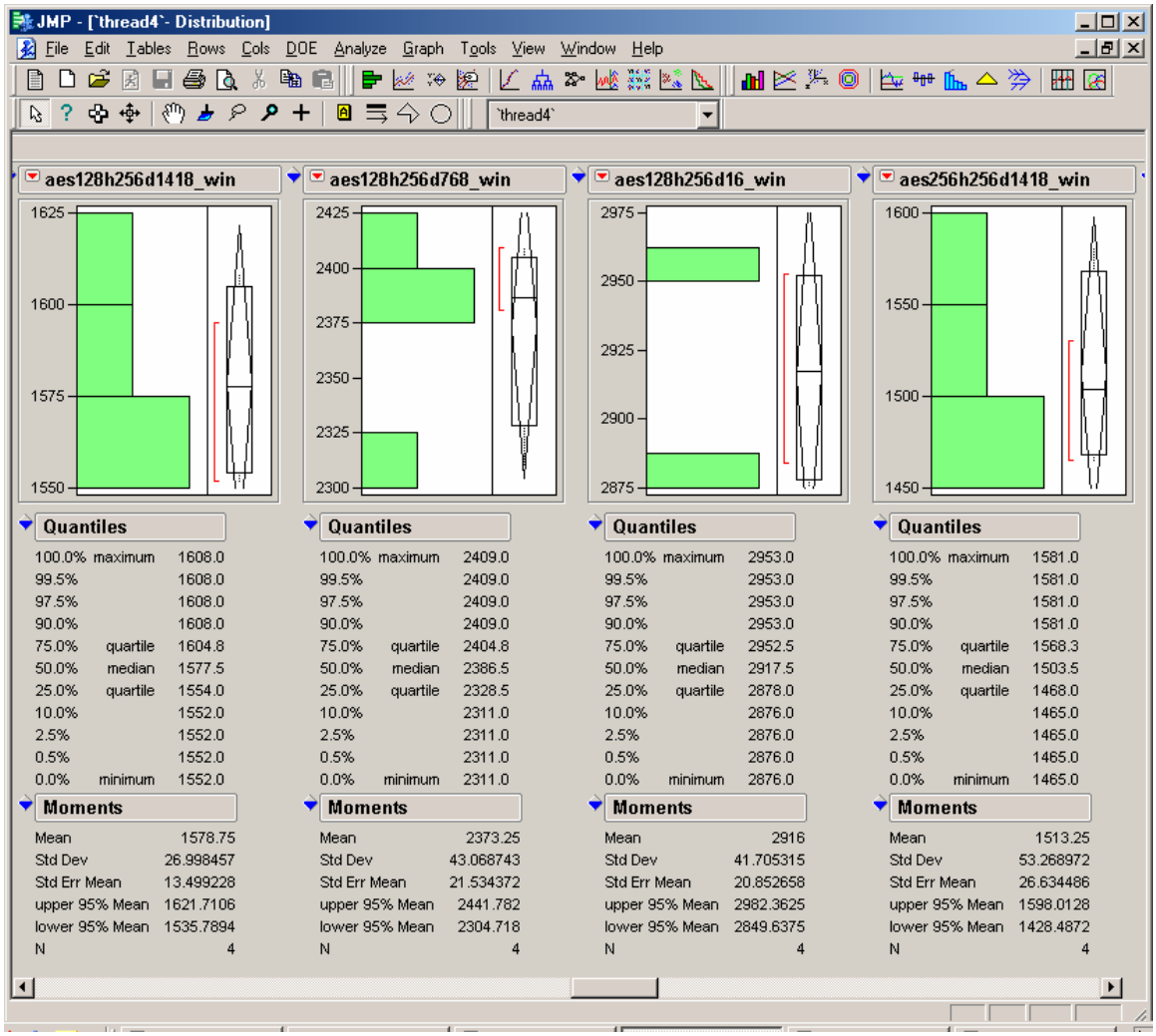

Figure A.7: JMP Distribution Data for Heap Size of 256 MB (Part 1)

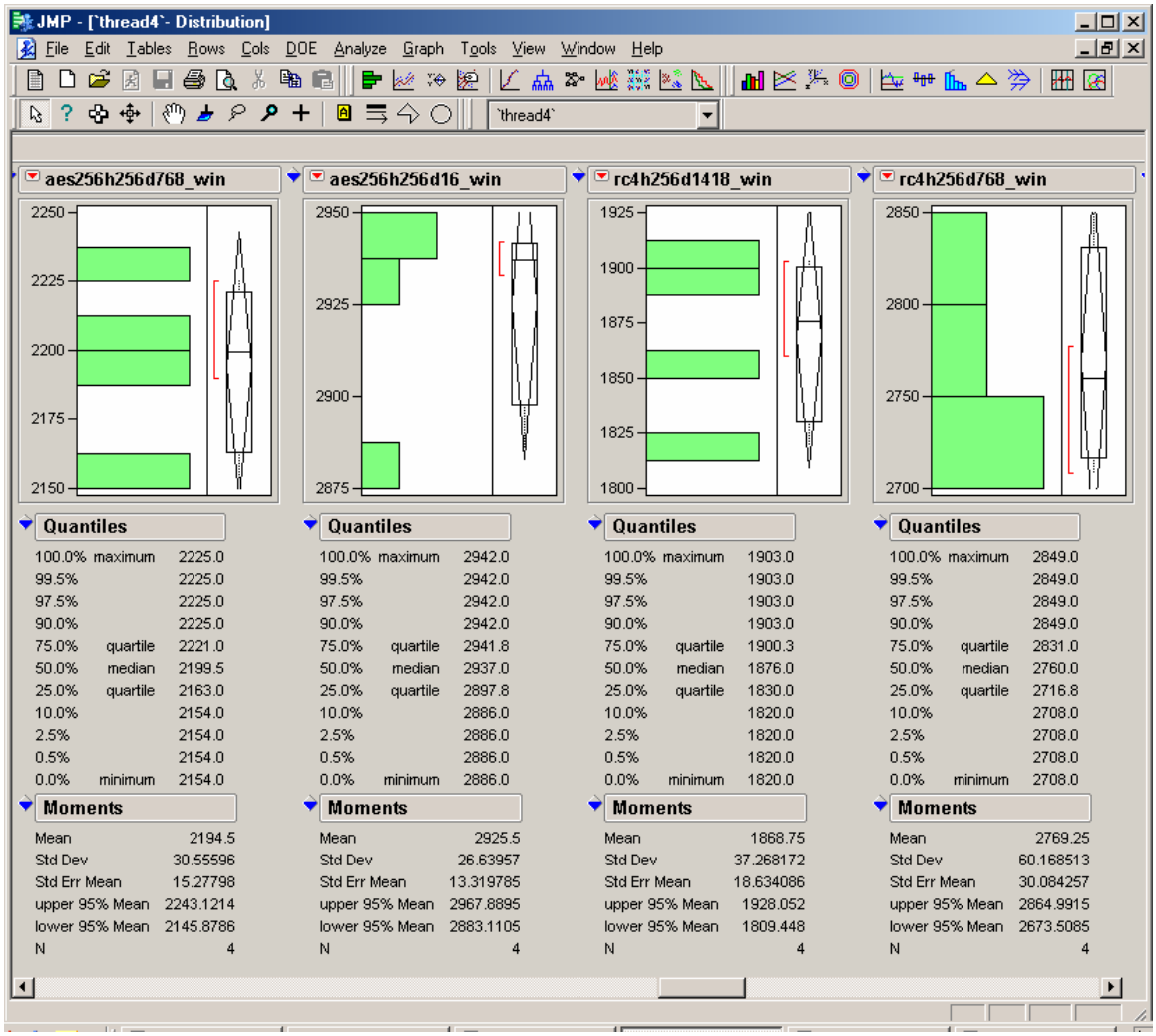

Figure A.8: JMP Distribution Data for Heap Size of 256 MB (Part 2)

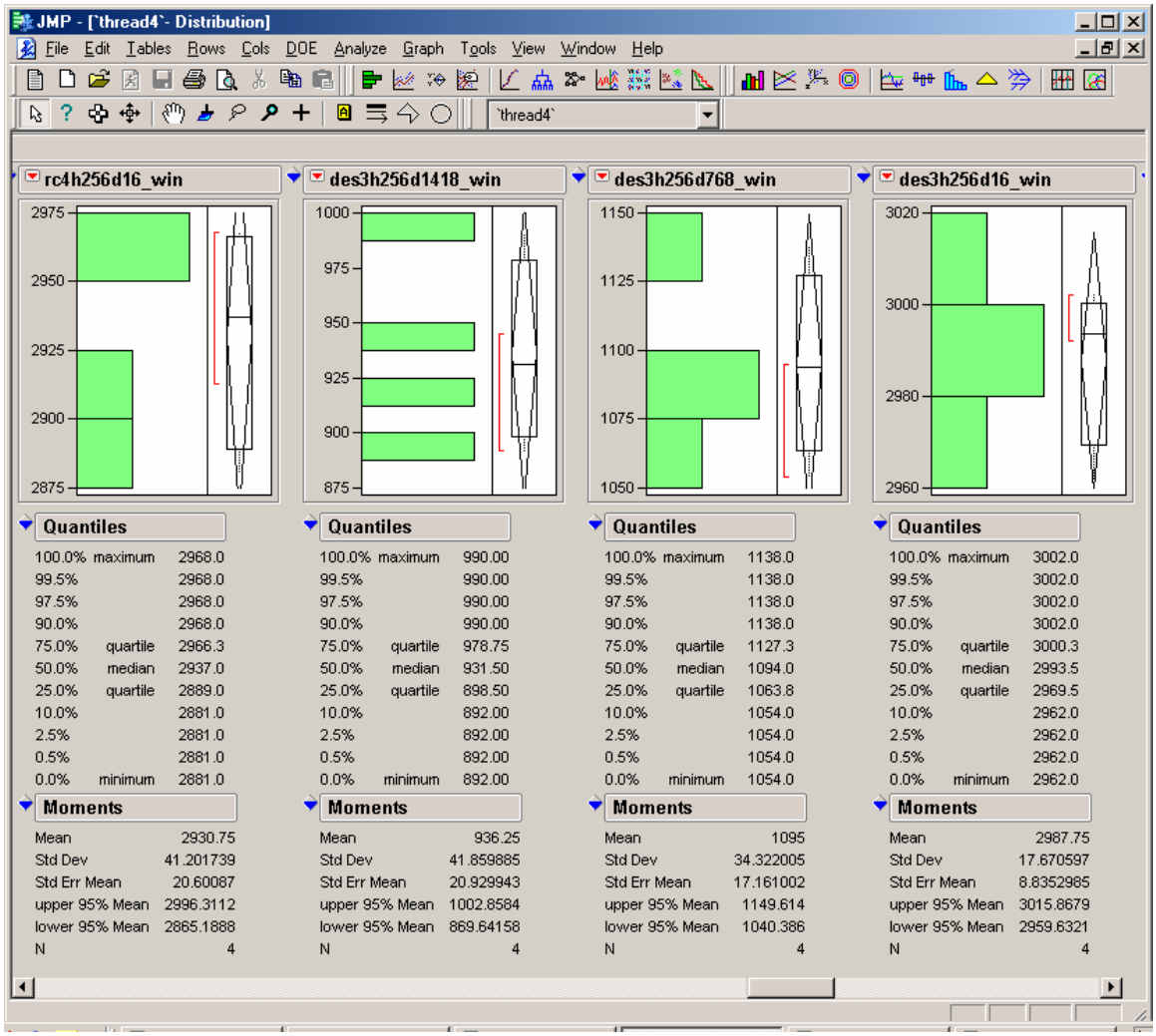

Figure A.9: JMP Distribution Data for Heap Size of 256 MB (Part 3)

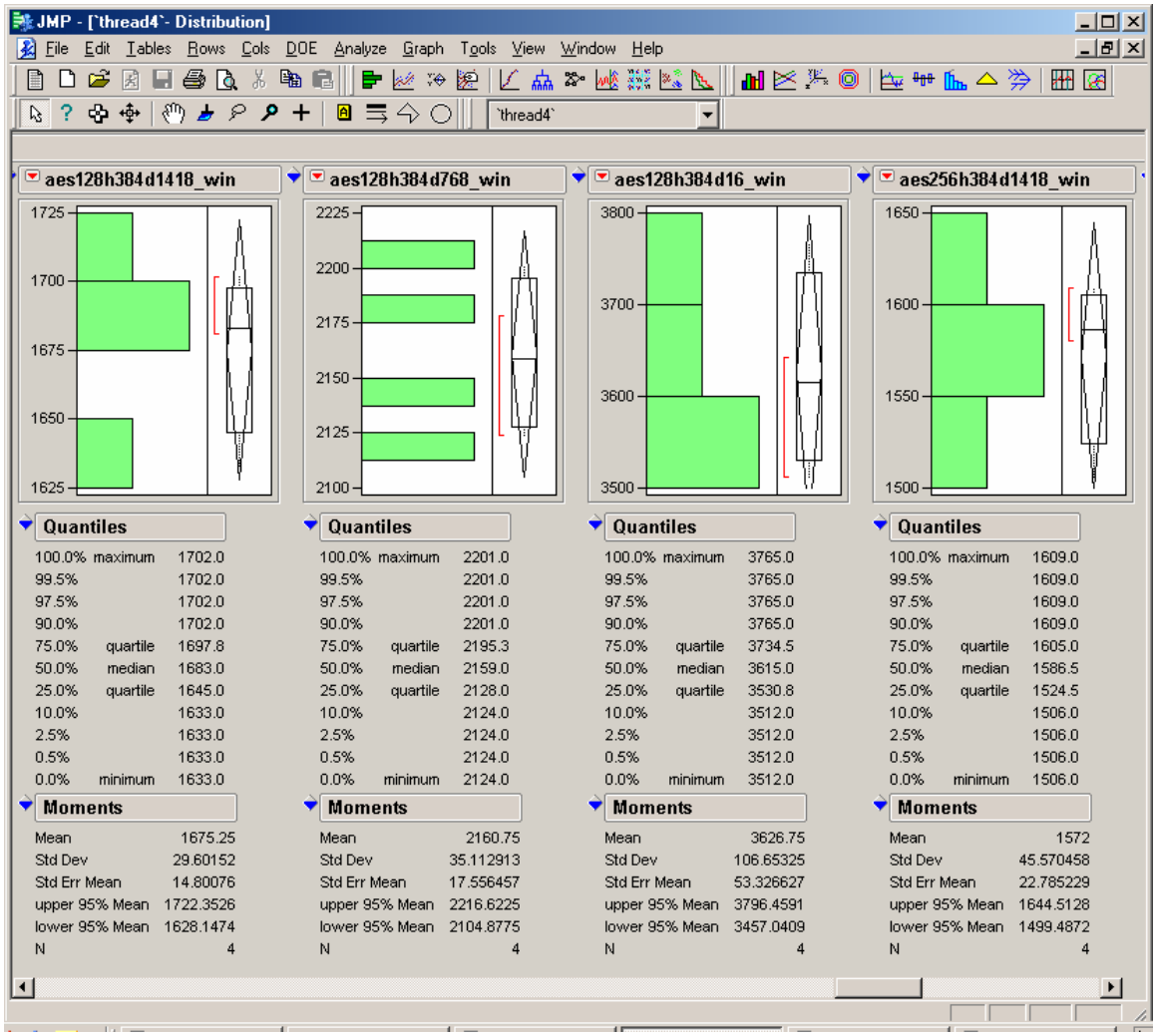

Figure A.10: JMP Distribution Data for Heap Size of 384 MB (Part 1)

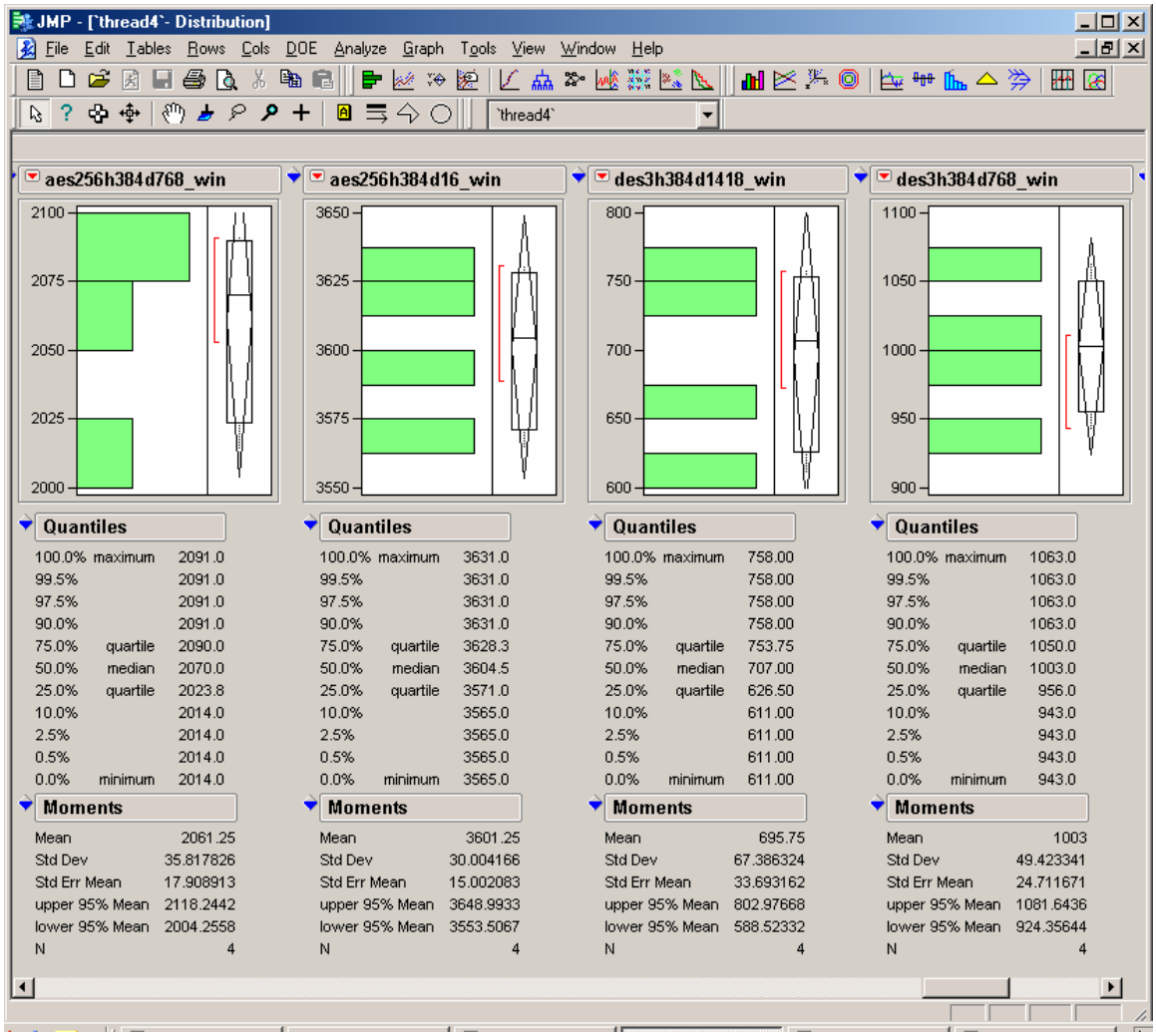

Figure A.11: JMP Distribution Data for Heap Size of 384 MB (Part 2)

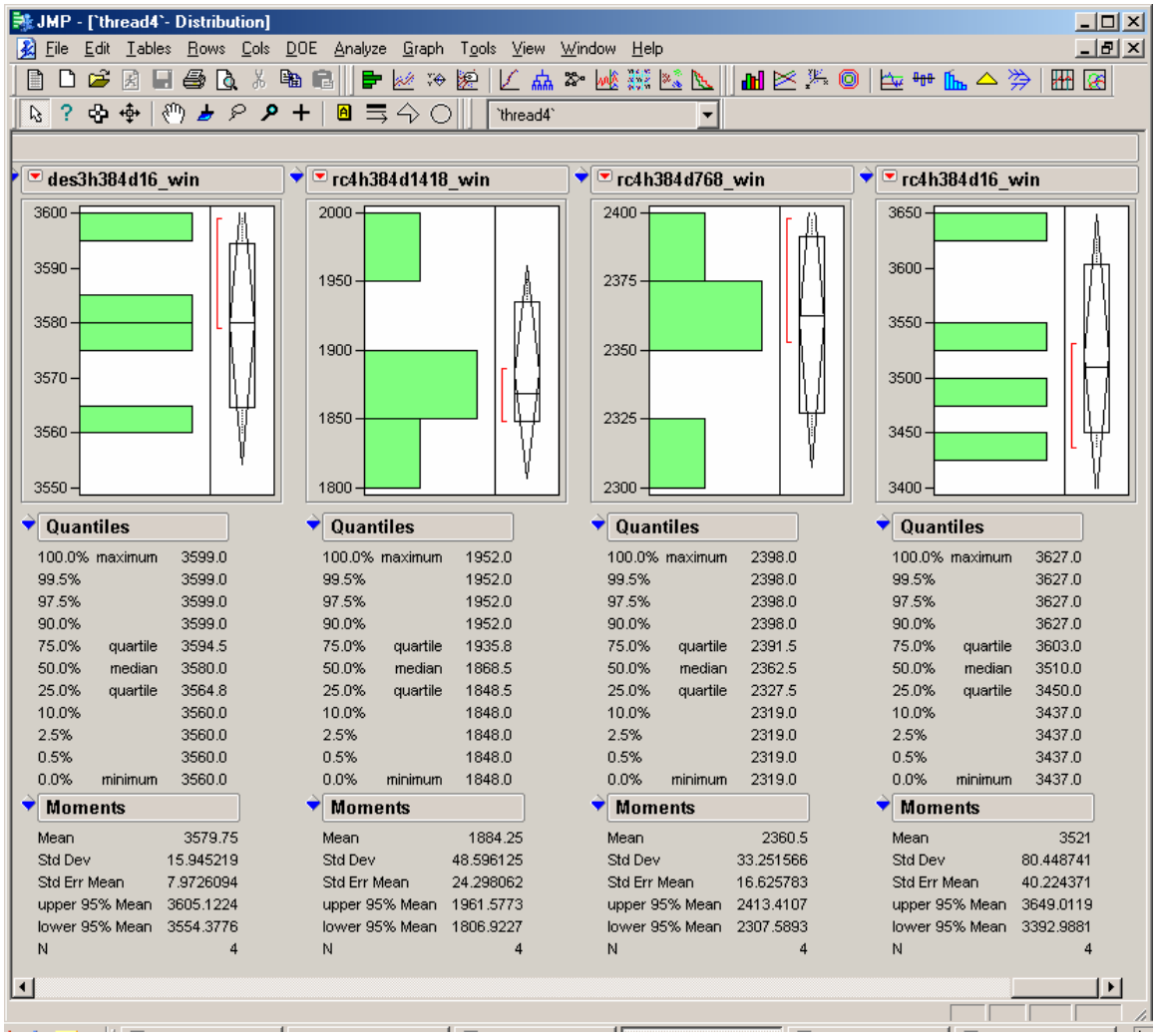

Figure A.12: JMP Distribution Data for Heap Size of 384 MB (Part 3)

Appendix B

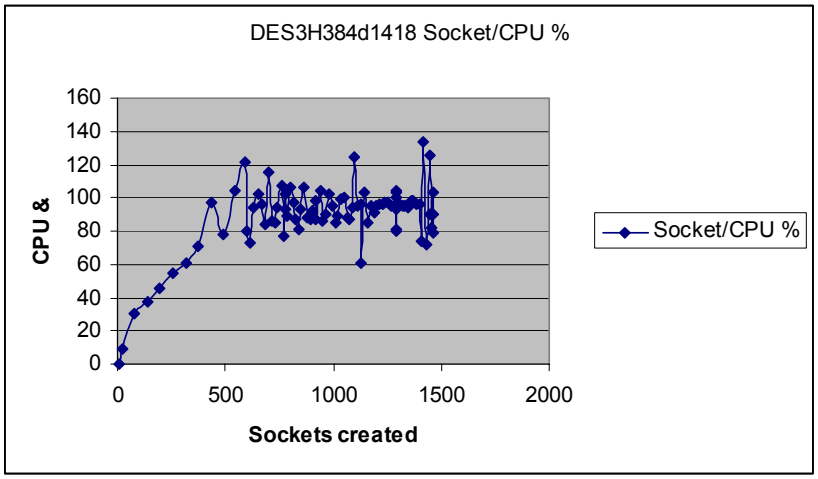

Figure B.1: CPU Utilization for DES3, Heap 384 MB, and Data Size 1418 Bytes

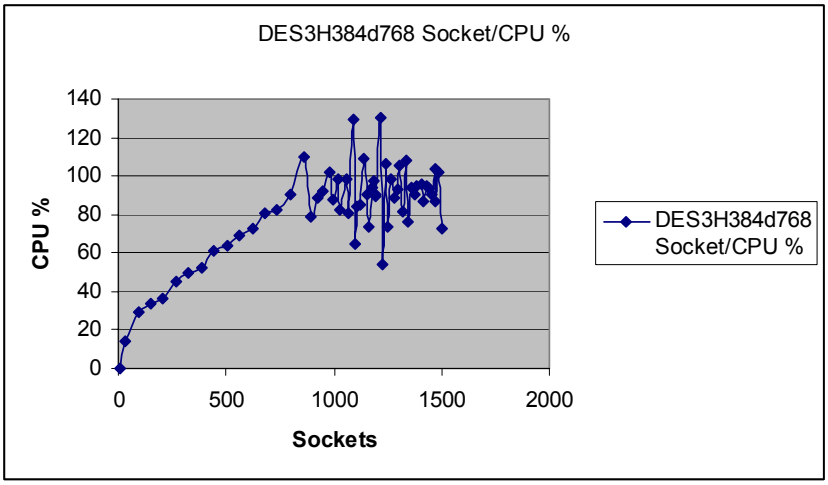

Figure B.2: CPU Utilization for DES3, Heap 384 MB, and Data Size 768 Bytes

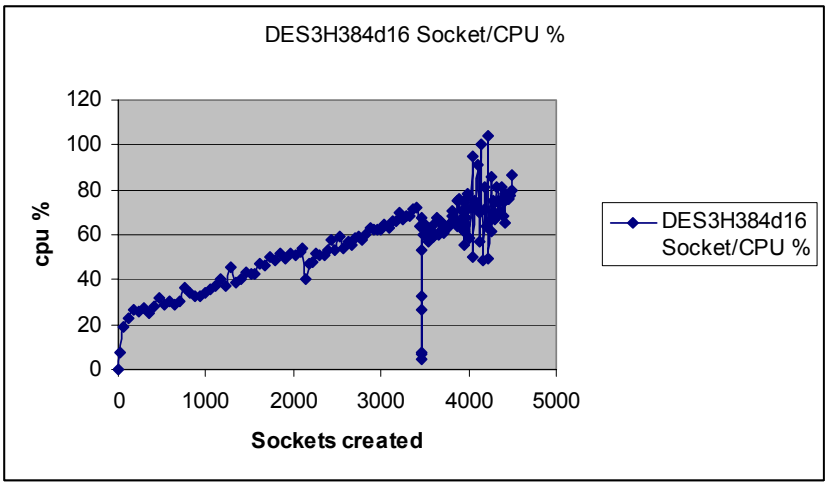

Figure B.3: CPU Utilization for DES3, Heap 384 MB, and Data Size 16 Bytes

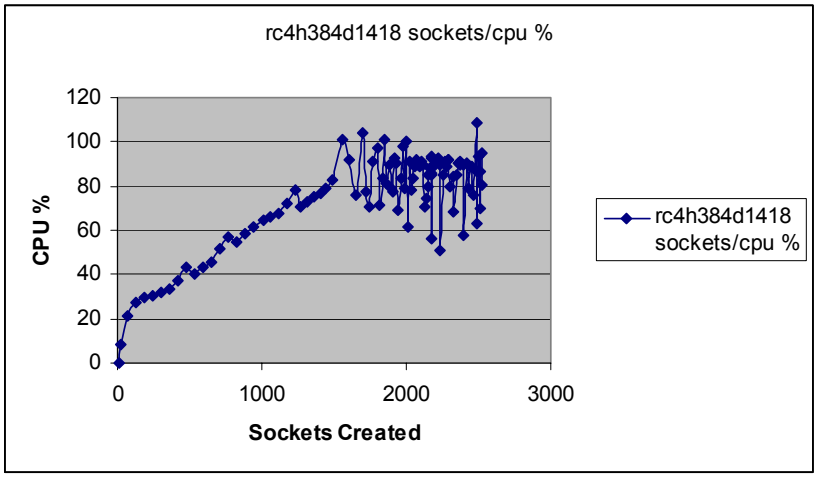

Figure B.4: CPU Utilization for RC4, Heap 384 MB, and Data Size 1418 Bytes

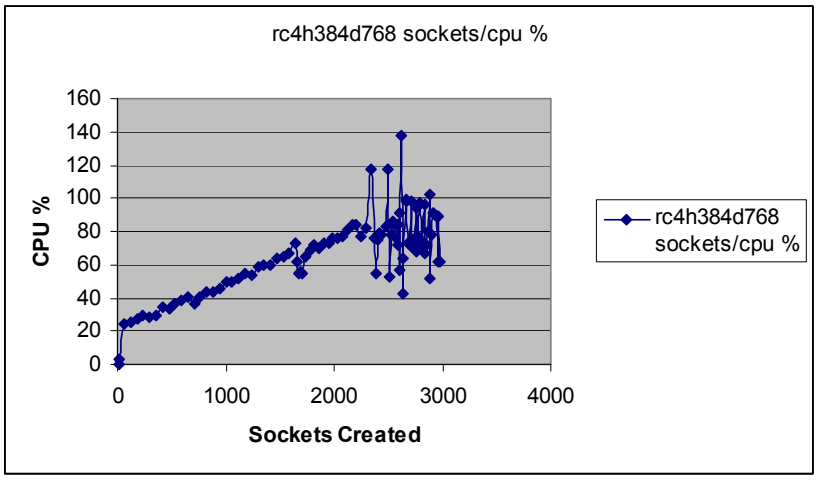

Figure B.5: CPU Utilization for RC4, Heap 384 MB, and Data Size 768 Bytes

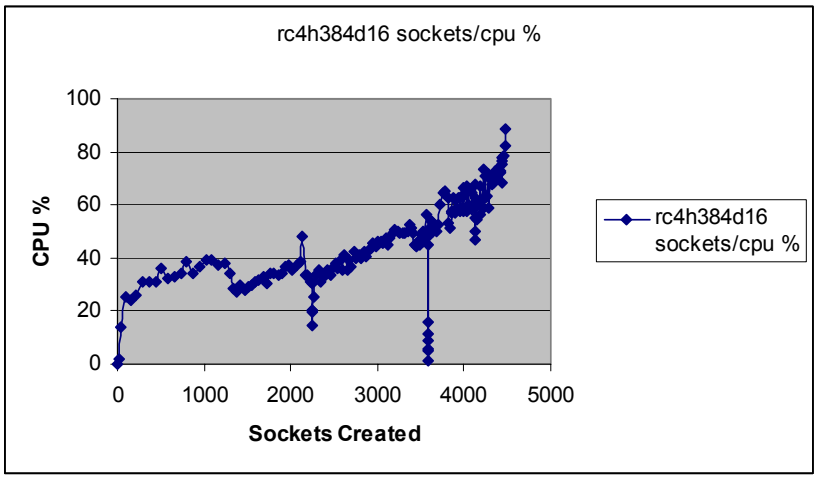

Figure B.6: CPU Utilization for RC4, Heap 384 MB, and Data Size 16 Bytes

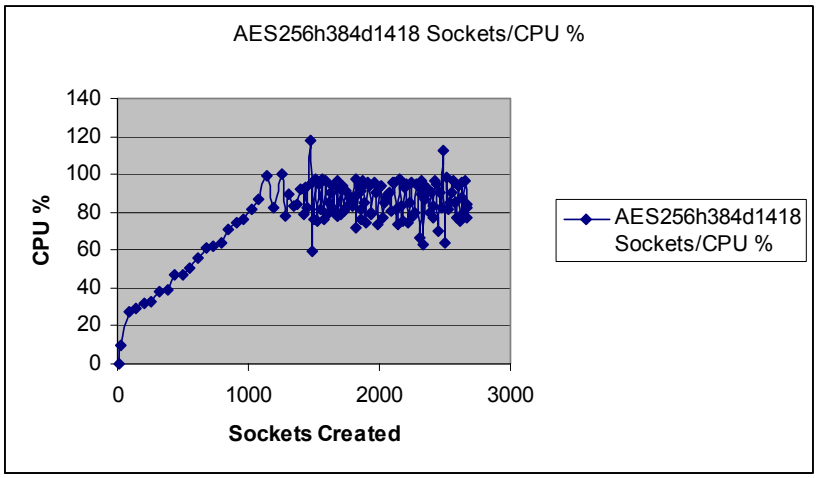

Figure B.7: CPU Utilization for AES 256-bit, Heap 384 MB, and Data Size 1418 Bytes

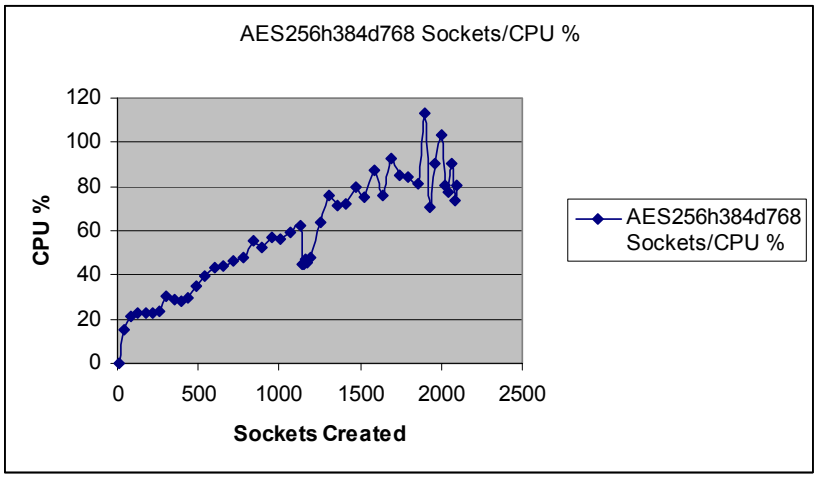

Figure B.8: CPU Utilization for AES 256-bit, Heap 384 MB, and Data Size 768 Bytes

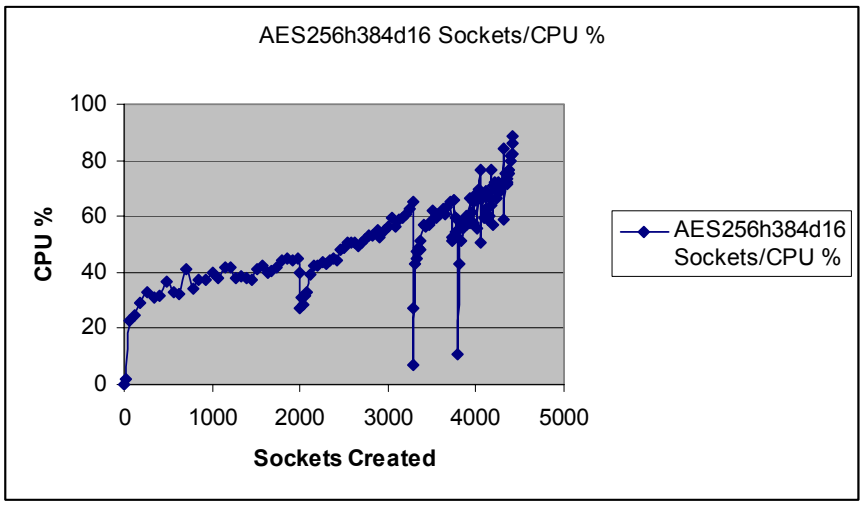

Figure B.9: CPU Utilization for AES 256-bit, Heap 384 MB, and Data Size 16 Bytes

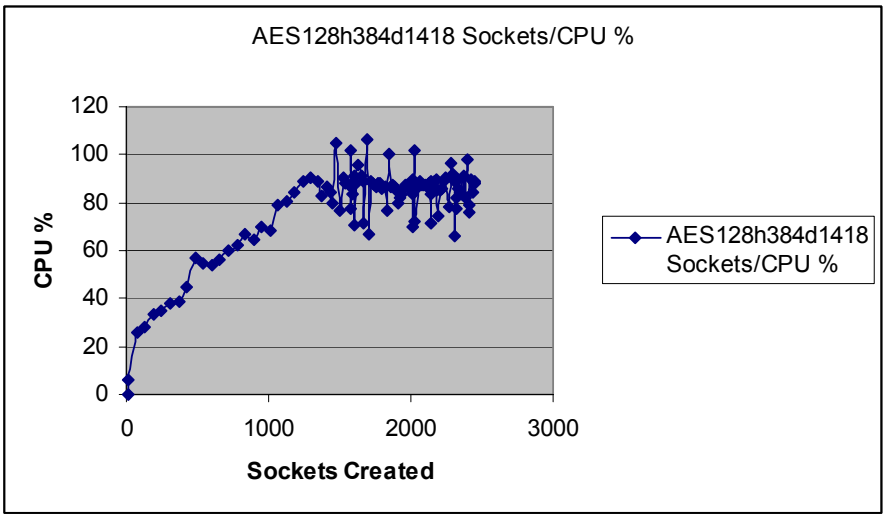

Figure B.10: CPU Utilization for AES 128-bit, Heap 384 MB, and Data Size 1418 Bytes

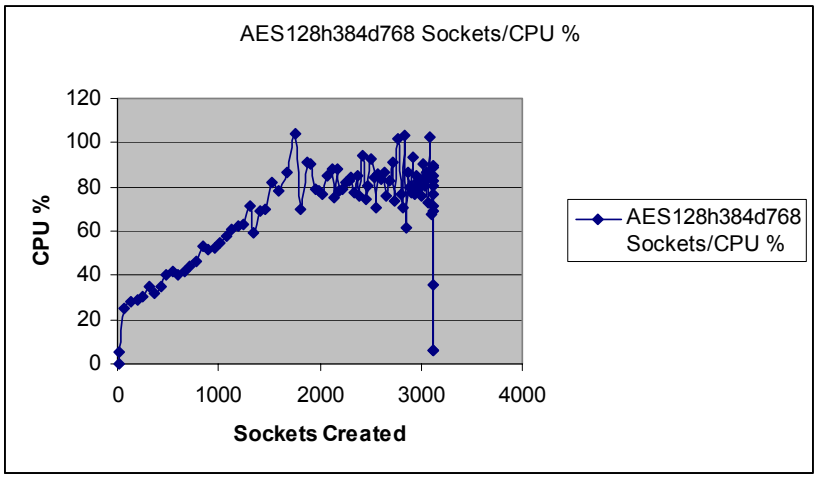

Figure B.11: CPU Utilization for AES 128-bit, Heap 384 MB, and Data Size 768 Bytes

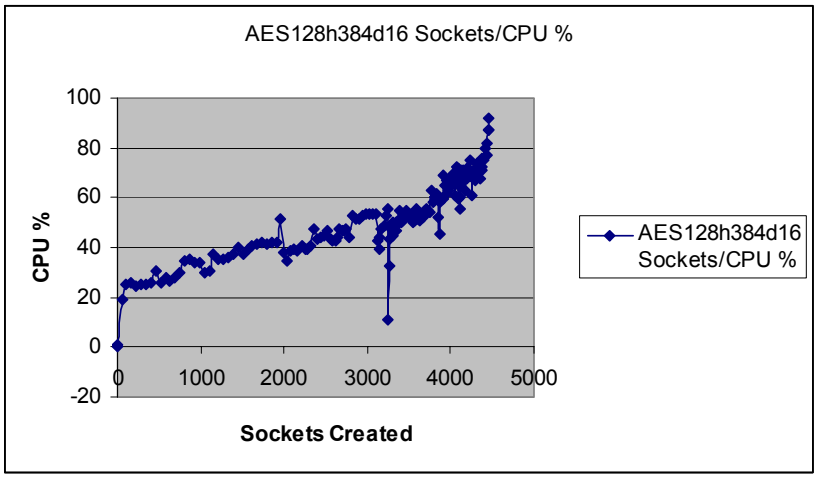

Figure B.12: CPU Utilization for AES 128-bit, Heap 384 MB, and Data Size 16 Bytes

Appendix C

KeyManagerFactory - This class acts as a factory for key managers based on a source of key material. Each key manager manages a specific type of key material for use by secure sockets. The key material is based on a KeyStore and/or provider specific sources [Sun03].

KeyManagerFactory::getDefault()-Generates a `KeyManagerFactory` object that implements the specified key management algorithm [Sun03].

KeyManager- Base interface for JSSE key managers. These manage the key material which is used to authenticate to the peer of a secure socket [Sun03].

SecureRandom- This class provides a cryptographically strong pseudo-random number generator (PRNG) [Sun03a].

SSLContext- Instances of this class represent a secure socket protocol implementation which acts as a factory for secure socket factories. This class is initialized with an optional set of key and trust managers and source of secure random bytes [Sun03].

SSLContext::getInstance()-Generates a `SSLContext` object that implements the specified secure socket protocol [Sun03].

SSLServerSocketFactory- This class creates SSL server sockets [Sun03].

SSLServerSocket- This class extends ServerSockets and provides secure server sockets using protocols such as the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocols. Instances of this class are generally created using a SSLServerSocketFactory. The primary function of SSLServerSockets is to create SSLSockets by accepting connections. SSLServerSockets contain several pieces of

state data which are inherited by the SSLSocket at socket creation. These include the enabled cipher suites and protocols, whether client authentication is necessary, and whether created sockets should begin handshaking in client or server mode. The state inherited by the created SSLSocket can be overriden by calling the appropriate methods [Sun03].

SSLSession – Public Interface - In SSL, sessions are used to describe an ongoing relationship between two entities. Each SSL connection involves one session at a time, but that session may be used on many connections between those entities, simultaneously or sequentially. The session used on a connection may also be replaced by a different session. Sessions are created, or rejoined, as part of the SSL handshaking protocol. Sessions may be invalidated due to policies affecting security or resource usage. Session management policies are typically used to tune performance [Sun03].

SSLSocketFactory- Instances of this kind of socket factory return SSL sockets. An SSL implementation may be established as the "default" factory [Sun03].

SSLSocket- SSLSocket is a class extended by sockets which support the "Secure Sockets Layer" (SSL) or IETF "Transport Layer Security" (TLS) protocols. Such sockets are normal stream sockets (java.net.Socket), but they add a layer of security protections over the underlying network transport protocol, such as TCP [Sun03].

TrustManagerFactory- The `javax.net.ssl.TrustManagerFactory` is an engine class for a provider-based service that acts as a factory for one or more types of `TrustManager` objects [Sun03].

TrustManagerFactory::getDefault()-Generates a `TrustManagerFactory` object that implements the specified trust management algorithm [Sun03].

TrustManager- The primary responsibility of the `TrustManager` is to determine whether the presented authentication credentials should be trusted. If the credentials are not trusted, the connection will be terminated. To authenticate the remote identity of a secure socket peer, you need to initialize an `SSLContext` object with one or more `TrustManager`s. You need to pass one `TrustManager` for each authentication mechanism that is supported. If null is passed into the `SSLContext` initialization, a trust manager will be created for you. Typically, there is a single trust manager that supports authentication based on X.509 public key certificates [Sun03].

Bibliography

[ApP00] Apostolopoulos, G., V. Peris, P. Pradhan, D. Saha, "Securing Electronic Commerce: Reducing the SSL Overhead," IEEE Network, 8-16 (July/August 2000).

[Bar01] Barber, R., "Hacking Techniques: The tools that hackers use, and how they are evolving to become more sophisticated," Computer Fraud & Security, Volume: 2001, Issue 3, pp. 9-12, (March 2001).

[BlD96] Blaze, M., W. Diffie, R.L. Rivest, B. Schneier, T. Shimo – Mura, E. Thompson, M. Wiener, "Minimal Key Length for Symmetric Ciphers to Provide Adequate Commercial Security," A Report by an Ad Hoc Group of Cryptographers and Computer Scientists, (January 1996).

[BoB03] Boneh, D., D. Brumley, "Remote Timing Attacks are Practical," White Paper from Stanford University, http://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf, (March 2003)

[Bon99] Boneh, D., "Twenty Years of Attacks on the RSA Cryptosystem", j-NAMS 46 number 2, pp. 203-213, (February 1999).

[Bur02] Burkholder, P., "SSL Man-In-The-Middle Attacks," SANS Infosec Reading Room, http://www.sans.org/rr/threats/man_in_the_middle.php, (February 2002).

[Cho02] Chou, W., "Inside SSL: The Secure Sockets Layer Protocol," IT Pro, 37-41 (September/October 2002).

[Cho02a] Chou, W., "Inside SSL: Accelerating Secure Transactions," IT Pro, 47-52 (July/August 2002).

[Gre02] Greenfield, D., "SSL and TLS", Network Magazine, http://www.networkmagazine.com/article/NMG20021203S0012/2, (December 2002).

[Har03] Harding, S., "Basic Cryptography, Part 7. One-Time Pads", The Sierra Times, http://www.sierratimes.com/03/09/03/science.htm, (March 2003).

[McG02] McGraw, G., "On Bricks and Walls: Why Building Secure Software is Hard," Computers and Security, Volume 21, Issue 3, pp. 229-238, (June 2002).

[McS03] McClure, S., Sa. Shah, Sh. Shah, "Web Hacking – Attacks and Defense", Pearson Education Inc., (2003).

[Nac99] Naccache, D., "Padding Attacks on RSA", Information Security Technical Report, Volume 4, Issue 4, pp. 28-33, (1999).

[NIST95] National Institute of Standards and Technology, "Secure Hash Standard", http://www.itl.nist.gov/fipspubs/fip180-1.htm, (April 1995).

[NIST98] National Institute of Standards and Technology, "SKIPJACK and KEA Algorithm Specifications Version 2.0", http://csrc.nist.gov/CryptoToolkit/skipjack/skipjack.pdf, (May 1998).

[NIST99] National Institute of Standards and Technology, "Federal Information Processing Standards Publication 46-3", http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf, (October 1999).

[NIST00] National Institute of Standards and Technology, "Federal Information Processing Standards Publication 186-2", http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf (January 2000).

[NIST01] National Institute of Standards and Technology, "Federal Information Processing Standards Publication 197", http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf, (November 2001).

[PeC00] Perrone, P.J., V. Chaganti, "Building Java Enterprise Systems with J2EE", Sam's Publishing (2000).

[Rei96] Reid, J., "Plugging the Holes in Host-based Authentication", Computers & Security, Volume 15, Issue 8, pp. 661-671, (1996).

[RFC1321] Request for Comment 1321, "The MD-5 Message Digest Algorithm", http://www.faqs.org/rfcs/rfc1321.html, (April 1992).

[RFC2246] Request for Comment 2246, "The TLS Protocol Version 1.0", http://ietf.org/rfc/rfc2246.txt, (January 1999).

[RFC2268] Request for Comment 2268, "A Description of the RC2(r) Encryption Algorithm", http://www.faqs.org/rfcs/rfc2268.html, (March 1998).

[RFC2437] Request for Comment 2437, "PKCS #1: RSA Cryptography Specifications Version 2.0", http://www.faqs.org/rfcs/rfc2437.html, (October 1998).

[Riv95] Rivest, R.L., "The RC5 Encryption Algorithm", CryptoBytes (1) 1 (Spring 1995).

[Sch95] Schneier, B., "The Blowfish Encryption Algorithm – One Year Later", Dr. Dobb's Journal, (September 1995).

[Sch98] Schneier, B., "Cryptographic design vulnerabilities," Computer, Volume 31, Issue 9, pp 29-33, (September 1998).

[Sun03] Sun Microsystems, “Java Secure Socket Extension (JSSE) 1.0.3 API User’s Guide,” http://java.sun.com/products/jsse, (May 2003).

[Sun03a] Sun Microsystems, “SecureRandom Class,” http://java.sun.com/j2se/1.4/docs/api/java/security/SecureRandom.html.

[Whi03] Whittaker, J., “How to Break Software: A Practical Guide to Testing”, Pearson Education (2003).

[ViM02] Viega, Messier, Chandra, “Network Security with OpenSSL,” O’Reilly (2002).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
23-03-2004	Master's Thesis	March 2003 – March 2004

4. TITLE AND SUBTITLE

AN ANALYSIS OF THE PERFORMANCE AND SECURITY OF J2SDK 1.4 JSSE IMPLEMENTATION OF SSL/TLS

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Bias, Danny R., Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Building 640
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/04-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Neal Ziring
TD/C4 NSA
Fort George G. Meade, MD 20755-6000
410-854-6191

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Java SSL/TLS package distributed with the J2SE 1.4.2 runtime is a Java implementation of the SSLv3 and TLSv1 protocols. Java-based web services and other systems deployed by the DoD will depend on this implementation to provide confidentiality, integrity, and authentication. Security and performance assessment of this implementation is critical given the proliferation of web services within DoD channels. This research assessed the performance of the J2SE 1.4.2 SSL and TLS implementations, paying particular attention to identifying performance limitations given a very secure configuration.

The performance metrics of this research were CPU utilization, network bandwidth, memory, and maximum number of secure socket that could be created given various factors. This research determined an integral performance relationship between the memory heap size and the encryption algorithm used. By changing the default heap size setting of the Java Virtual Machine from 64 MB to 256 MB and using the symmetric encryption algorithm of AES256, a high performance, highly secure SSL configuration is achievable. This configuration can support over 2000 simultaneous secure sockets with various encrypted data sizes. This yields a 200 percent increase in performance over the default configuration, while providing the additional security of 256-bit symmetric key encryption to the application data.

15. SUBJECT TERMS

Java Programming Language; Performance(Engineering); Test and Evaluation; Electronic Security; Cryptography; Secure Communications

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Richard A. Raines, AD-23, DAF
U	U	U	UU	98	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-6565, ext 4278 (richard.raines@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18